EFFICIENT SHUFFLE FOR FLASH BURST COMPUTING

A DISSERTATION
SUBMITTED TO THE DEPARTMENT OF COMPUTER SCIENCE
AND THE COMMITTEE ON GRADUATE STUDIES
OF STANFORD UNIVERSITY
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY

Yilong Li
December 2022

I certify that I have read this dissertation and that, in my opinion, it is fully adequate in scope and quality as a dissertation for the degree of Doctor of Philosophy.

John Ousterhout, Primary Adviser

I certify that I have read this dissertation and that, in my opinion, it is fully adequate in scope and quality as a dissertation for the degree of Doctor of Philosophy.

Keith Winstein

I certify that I have read this dissertation and that, in my opinion, it is fully adequate in scope and quality as a dissertation for the degree of Doctor of Philosophy.

Matei Zaharia

Approved for the Stanford University Committee on Graduate Studies.

Stacey F. Bent, Vice Provost for Graduate Education

This signature page was generated electronically upon submission of this dissertation in electronic format.

Abstract

Shuffle is the operation of exchanging arbitrary data among a group of servers, and it is a fundamental communication primitive in distributed computing. In particular, shuffle has been shown to be a fundamental bottleneck to the scalability of flash bursts, a radically new paradigm in datacenter computing. Flash bursts use a large number of servers but for very short time intervals (as little as one millisecond), which makes it possible to run large-scale data-intensive computation within a few milliseconds.

The unique features of flash bursts present three significant challenges to the design of a shuffle algorithm. First, to avoid creating a central bottleneck, it's not feasible to run a centralized scheduler. Second, flash bursts will likely be colocated with traditional batch jobs to achieve high resource efficiency, so they need to perform shuffles efficiently even under interference from competing workloads. Finally, when running at large scales, the underlying network can rarely provide full bisection bandwidth that is critical to most existing algorithms.

This thesis describes an efficient shuffle algorithm designed from clean slate that achieves optimal performance cross a wide range of workloads even under the demanding conditions of flash bursts. This algorithm uses a local policy at end hosts to schedule its incoming and outgoing messages as ensembles in order to match sender and receiver bandwidths in a decentralized way; it overcommits the receiver downlinks to maintain high network utilization under interference; it uses pro rata sliding windows in a receiver-driven rate control scheme to implement weighted max-min fair sharing of the downlink capacity; finally, when the underlying network doesn't provide full bisection bandwidth, it uses a lightweight global scheduling scheme to manage the bottleneck links in the core of the network without requiring a central scheduler or the knowledge of the underlying network topology. The evaluation shows that the throughput of the resulting shuffle algorithm is usually within a few percent of the theoretical limit even in the 90th percentile.

Acknowledgments

First and foremost, I would like to thank my advisor, John Ousterhout, for his support and guidance during my PhD journey. I have learned so many things from John over the years, from designing and writing highly performant systems, to pushing the envelop of system research by asking questions relentlessly, and to presenting complex research concepts in a clear way. John is the most devoted educator I have ever seen and the best advisor I could've wished for. He has made my long journey of PhD well worth it, and now I am excited to apply everything I have learned to my next challenge and pass on the knowledge to the people around me.

Also thank you to my committee members, Keith Winstein, Matei Zaharia, Christos Kozyrakis, and Mac Schwager for both providing feedback on my work and attending my thesis defense. Special thanks to Keith Winstein and Matei Zaharia for helping me both shape the thesis topic and guide me through the literature that allows me to position the work properly.

Thanks to my labmates, Ankita Kejriwal, Behnam Montazeri, Henry Qin, Jacqueline Speiser, Seo Jin Park, Sarah Tollman, Stephen Yang, Jonathan Ellithorpe, and Collin Lee, for their help and support in everything. Special thanks to Seo Jin Park and Collin Lee for the fruitful discussions and collaboration along the way.

I also must thank my undergraduate professors at UIUC, Darko Marinov and Grigore Rosu, and senior PhD students, Milos Gligoric and Andrei Stefanescu, for introducing me to academic research in software engineering and formal methods, which eventually led me down the path of pursuing a PhD. Darko and Grigore also helped me immensely with my graduate school applications.

Finally, I am deeply grateful to my parents, Jianping Li and Zhaoping Wang, and my wife, Yu Qin, for their unconditional love, patience, and encouragement. Without my parents' foresight and support, I would never be able to come to the US for college or join the prestigious PhD program at Stanford. And my wife has been a persistent source of encouragement and joy for me along the journey, and I owe her many thanks for listening to my ramblings, and sharing in my successes and failures.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Shuffle is the operation of exchanging arbitrary data among a group of servers, and it is a fundamental communication primitive in distributed computing. For example, it is widely used in the areas of large-scale data processing (e.g., distributed sorts and joins), distributed graph analytics, and HPC simulations.

However, there has been relatively little research on efficient shuffle algorithms. I suspect this is due to at least three reasons. First, shuffles in existing applications usually have regular patterns that are less challenging. For example, in distributed sorts, every server often sends and receives roughly the same amount of data, and the shuffle messages are roughly the same size as well. Second, shuffles require transmitting every byte of the data over the network at least once, so there is no room for bandwidth-saving optimizations. Third, shuffles are actually not the bottleneck in many workloads, so people have little interest in optimizing shuffle algorithms.

But why do we need a new shuffle algorithm now? The motivation of this thesis is primarily driven by the recent emergence of a radically new paradigm in datacenter computing called *flash bursts*. Flash burst applications use a large number of servers but for very short time intervals (as little as one millisecond). Thus, it is possible to run large-scale data-intensive computation within a few milliseconds with flash bursts, which may unlock new exciting applications such as complex real-time decision making algorithms. In contrast, today's datacenter applications couple scale and time: applications that harness large numbers of servers also execute for long periods of time (seconds or more).

Flash bursts call for a redesign of the shuffle algorithm for two reasons. First, the unique features of flash bursts present significant challenges to existing algorithms. For example, the combination of large number of servers and short time intervals makes it impossible to employ any scheme that runs the scheduling algorithm on a central server; flash bursts will likely need to be colocated with traditional batch workloads to improve resource efficiency, so they need to perform shuffles efficiently even under interference from competing jobs; finally, when running at large scales, the underlying

network can rarely provide full bisection bandwidth that is critical to most algorithms. Second, efficient flash bursts need efficient shuffles. As we shall see in Chapter 2, shuffle is almost always the critical bottleneck at the optimal operating points of flash bursts. To see why this is true, consider a flash burst application where the bottleneck is the local computation instead: this application can continue to harness more servers (and distribute its data in finer granularity) to speed up the computation until the overheads of coordination among servers (usually in the form of shuffles) outweigh the benefits of adding more servers. Thus, improving shuffle efficiency also improves the scalability of flash bursts.

As a result, this thesis sets out to design a new shuffle algorithm with a clean slate to solve the challenge of performing efficient shuffles in flash bursts once and for all. An ideal shuffle algorithm for flash bursts should have the following properties:

- It should scale smoothly to a large number of nodes (e.g., 1000 or more nodes).
- It should remain efficient even when the amount of data to shuffle is small (e.g., the shuffle may complete in less than one millisecond).
- It should be robust to interference from the competing workloads running in the background.
- It should schedule data transfers wisely to better utilize the core bandwidth of the network when the underlying network doesn't provide full bisection bandwidth.

Unfortunately, existing solutions are far from satisfying: they may be optimized only for very specific workloads, not applicable to the stringent time budgets of flash bursts, or suffering from significant performance degradation in non-ideal environments. For example, Weighted Shuffle Scheduling [14] (WSS) is a well-known algorithm that is provably optimal (under certain assumptions); however, its typical implementations are often not suitable to low-latency shuffles in flash bursts.

This thesis is structured into two parts. The first part of the thesis demonstrates the importance of shuffles by introducing the notion of flash bursts. In order to learn more about the feasibility of flash bursts, I developed two new benchmarks, MilliSort and MilliQuery. MilliSort is a distributed sorting application and MilliQuery implements three SQL queries. The goal for both applications was to process as many records as possible in one millisecond, given unlimited resources in a datacenter. The main contributions of this part are as follows:

- It proposes a radically new style of computation named flash burst that harnesses a large number of servers but for very short time intervals.
- It confirms that flash bursts are not just feasible for challenging workloads, such as sorting, that require complex coordination among servers, but also rather efficient even compared to big data systems that are designed to operate at much larger timescales.

- It yields several interesting insights about flash bursts: e.g., the amount of data that a flash burst can handle grows quadratically with the time budget (both the amount of data per server and the number of servers grow at least linearly with the time budget).
- It identifies that some of the most important and common limits to scalability are coordination overhead and shuffle cost (both can be largely attributed to per-message overheads, although performing efficient shuffles requires a lot more than just reducing per-message overheads).

The second part, which is also the main body of the thesis, focuses on how to perform shuffles as fast as possible, and the main contribution is a clean-slate shuffle algorithm that combines a number of simple yet effective ideas in a synergistic way:

- It advocates an ensemble-wide scheduling approach on end hosts to match sender and receiver bandwidths in a decentralized fashion: in particular, each sender uses a local scheduling policy, Greatest Remaining Processing Fraction (GRPF), to allocate its uplink bandwidth in proportion to the remaining sizes of the outgoing messages, and each receiver also uses GRPF to allocate its downlink bandwidth by issuing grants to senders.
- It discovers that overcommitting the receiver downlinks is the key to maintaining high network utilization, especially when there is interference from competing workloads.
- It uses a receiver-driven window-based rate control scheme to complement the GRPF policy in the presence of overcommitment; this scheme effectively prevents certain senders from obtaining unfairly large shares of downlink bandwidth, and it achieves straggler mitigation naturally by gradually assigning more bandwidth to the messages that fall behind.
- When a purely endhost-based solution falls short due to in-network bottlenecks, a simple yet effective heuristic is devised to mitigate straggler receivers arise from sub-optimal allocation of bandwidth in the network core; this heuristic is completely agnostic to the underlying network topology and only requires exchanging a small piece of data periodically among the cluster.

The net result is a shuffle algorithm can achieve excellent performance (usually within a few percent of the optimal) for a wide range of workloads and under various demanding conditions. In fact, this algorithm could be viewed as an instantiation of the optimal WSS scheme which better exploits the specifities of the low-latency shuffle problem (Chapter 3 will discuss their relationship in detail).

Note that the study on flash bursts preceded the work on efficient shuffles. I did not choose the example applications of flash bursts because they require shuffles, and I did not know about the limiting factors. However, it turned out that shuffle is not only essential in flash bursts (three out of four workloads require at least one shuffle), but also the critical limiting factor of the scalability of flash bursts. As a result, the first part of the thesis (Chapter 2) is self-contained, and it can be read independently from the other chapters.

The remainder of this dissertation describes the MilliSort and MilliQuery experiments for flash bursts (Chapter 2); explains the motivations, design rationales, and details of the new shuffle protocol (Chapter 3); covers the network simulator (Chapter 4) and workload generator (Chapter 5) that are used in the performance measurements; conducts a thorough evaluation of the shuffle protocol (Chapter 6); and discusses additional topics and related work (Chapter 7).

Chapter 2

MilliSort and MilliQuery

One of the benefits of datacenter computing is the ability to run large-scale applications that harness hundreds or thousands of machines working together on a common task. Using frameworks such as MapReduce [21] and Spark [100], developers can easily create applications in a variety of areas such as large-scale data analytics.

Until recently, large-scale applications have executed for relatively long periods of time: seconds or minutes. This was necessary to amortize the high cost of allocating and coordinating a collection of servers. Similarly, frameworks such as MapReduce and Spark have traditionally operated on very large blocks of data, in order to amortize high network latencies. Thus, they cannot be applied to real-time tasks. Instead, real-time queries must return precomputed results, such as those produced in overnight batch runs. This means that the queries must be carefully planned in advance; ad-hoc queries cannot easily be supported.

Streaming frameworks such as Flink [26] or Spark Streaming [101] operate on incoming data in real time, but they do this by incorporating new data into queries or transformations that are planned long in advance. In order to support real-time queries, the scale of computation triggered by each event is quite limited.

In recent years, new serverless platforms such as [5, 67, 31] have made it possible to run short-lived tasks (as small as a few hundred milliseconds) in datacenters. However, the unit of execution in these environments is an individual function call. It is difficult to harness multiple machines in a single serverless computation; for example, direct communication between lambdas is not officially supported, and existing workarounds [3, 27] have high latency and low bandwidth.

In this thesis, I set out to explore the possibility of extending serverless computing in two ways: first, by further reducing the timescale; and second, by reintroducing scale, so that large numbers of servers can work together. The term *flash burst* is used to describe a computation that has a very short lifetime yet harnesses large numbers of servers. Flash bursts offer the potential of analyzing large amounts of data in real time. This could enable the creation of new applications that execute

customized queries on large datasets in real time, without the need to predict queries hours or days ahead of time.

Rather than making incremental improvements on existing systems, my goal is to push the notion of flash bursts to the extreme, in order to understand the limits of this style of computation. In particular, I set out to answer the following questions:

- What is the smallest possible timescale at which meaningful flash bursts can operate?
- What is the largest number of servers that can be harnessed at such a timescale?
- What aspects of current systems limit the duration and scale of flash bursts?

We hypothesized timescales as small as 1 ms might be possible, so we set that as the initial goal.

In order to make the problem more concrete, I decided to implement two focused applications that capture patterns of computation and communication I expect to be common in flash bursts. The first application is MilliSort: given unlimited resources in a datacenter, what is the largest number of small records that can be sorted in one millisecond? The second application is MilliQuery, which consists of three representative SQL queries from the tutorials for Google BigQuery. The queries range from a simple scan-filter-aggregate query to a distributed join requiring multiple shuffles. As with MilliSort, the goal was to understand how much data can be analyzed, and how many servers can be harnessed, in timescales around 1 ms.

The process of developing and measuring these applications has yielded interesting results in three categories:

- Measurements: MilliSort and MilliQuery demonstrate that large-scale data analytics can operate efficiently even at timescales of 1–10 ms. MilliSort can sort 0.84 million small records in one millisecond using 120 servers running on 30 machines. The MilliQuery benchmarks process .03–48 million records in one millisecond using 60–280 servers, depending on the query.
- Observations: the development of MilliSort and MilliQuery yielded several interesting insights about flash bursts, which are summarized in Section 2.6. For example, I found that the amount of data that a flash burst can handle grows quadratically with the time budget (both the amount of data per server and the number of servers grow at least linearly with the time budget). Some of the most important and common limits to scalability are shuffle cost and coordination overhead (both can be attributed to per-message overheads).
- Algorithms: while implementing MilliSort we developed a new low-latency algorithm for partitioning the keys, which uses a hierarchical series of distributed sorts. I also developed a novel splitter selection algorithm that improves the balance among data partitions. Overall, MilliSort runs with efficiency comparable to other systems that operate at much larger timescales.

- Sort 40,000 10-byte keys using 8 cores [7].
- Copy 5 Mbytes of data from memory to memory sequentially.
- Send or receive 5 Mbytes of data with a 40 Gbps NIC.
- Invoke 300 back-to-back remote procedure calls on one core, using kernel bypass [77, 81, 50].
- Send or receive 2–5k small messages on one core [77, 50, 75].
- Take 10,000 back-to-back L3 cache misses on one core.

Figure 2.1: Examples of tasks that can be completed in one millisecond on modern hardware.

2.1 Background

One millisecond is not very long. Figure 2.1 lists a few things that can be done in one millisecond on today's machines; these create fundamental limitations for flash bursts.

2.1.1 Limited data per server

One of the most important limitations evident from Figure 2.1 is that each server can only manipulate a small amount of data (on the order of a few Mbytes): e.g., a single server core can only access a few megabytes of data in one millisecond, and network bandwidth allows only a few megabytes to be transmitted in one millisecond.

Given the large number of servers and small data per server, data must stay evenly distributed throughout a flash burst. If even a small fraction of data accumulates on a single server, the network link into that server will become a bottleneck.

2.1.2 Coordination cost

Given that each server can only access a small amount of data, the overall scale of a flash burst will be limited by the number of servers that can be harnessed. But, the small time scale makes it difficult to coordinate very many servers; at some scale one millisecond isn't even enough time to notify all the servers to start working. Thus, coordination overheads play a fundamental role in flash bursts, since they limit the scale. "Coordination" includes such activities as engaging all of the servers, determining work assignments for each server, and sequencing the phases of the algorithm. Existing large-scale applications such as Spark store much larger amounts of data per server and also run for longer time periods; this combination makes coordination overheads less important.

2.1.3 Multiple communication costs

For many existing large-scale applications, the only communication cost that matters is network bandwidth. Systems such as MapReduce and Spark are explicitly designed to exploit bandwidth and hide communication latency. However, for flash bursts three different costs may become important.

In addition to *bandwidth*, which matters when sending large blocks of data, and *latency*, which matters when sending small chunks of data, a third cost plays an important role in flash bursts: *per-message overhead* (the CPU time required to send and receive short messages). Per-message overhead comes into play when a server has a collection of small requests that can be sent to other servers concurrently; it limits how quickly a series of messages can be issued or received. Per-message overheads are particularly important in flash bursts because they dominate the cost of group communication primitives (discussed below), which in turn dominate the cost of coordination.

2.1.4 Group communication

If a collection of servers is to cooperate closely, the servers will probably need to exchange data frequently and in small chunks. However, in a flash burst, where there are hundreds or thousands of servers operating on a very small time scale, it is not practical for each server to communicate directly with all of the other servers. For example, if a server broadcasts data to 1000 other servers by sending a separate small message to each of them, the broadcast will consume a substantial fraction of a millisecond, due to per-message overheads.

Thus, *group communication* plays an important role in flash bursts. In group communication, many or all of the servers in a cluster transmit data concurrently to carry out a cluster-wide goal. The HPC community has identified and implemented a variety of useful group communication mechanisms [90], of which four play a role in MilliSort and MilliQuery:

- **Broadcast:** data that is initially present on a single server must be distributed to every server in the group.
- **Gather:** the reverse of broadcast. A single server must collect distinct data from each of the servers in the group.
- **All-gather:** initially each server in the group stores a distinct data item; the all-gather operation must arrange for every server in the group to receive a copy of all the items.
- **Shuffle:** each server initially stores a separate data item for every other server in the group; the shuffle must transmit all the items to their intended targets.

Group communication provides two benefits. First, it harnesses multiple servers operating concurrently to speed up the communication; for example, several servers can be used to complete a broadcast more quickly by propagating messages via a tree structure. Second, it can sometimes replace many small messages with a few larger messages; this reduces the impact of per-message overheads. For example, a hypercube style [93] of all-gather requires only $MlogM$ messages for M servers, vs. M^2 messages if each server communicates independently with every other server.

2.2 Applications

One of the challenges in exploring flash bursts is that there are no flash burst applications available today (unsurprising, since there is no infrastructure capable of supporting them). Fortunately, there appears to be a small set of patterns of computation and communication that are used commonly across a variety of large-scale applications and account for much of their performance [4, 30, 51]. These are referred to as *dwarfs* by Asanovic et al. [4] and others. For example, matrix operations, sorting, and statistics computations are dwarfs for big-data and AI workloads. Rather than guessing at full applications, I have implemented two small applications that capture several dwarfs. Although the behavior of these dwarfs will not be a perfect predictor of any real application, this approach has two advantages. First, lessons learned from the dwarfs are likely to apply to many real applications. Second, it is easier to understand the properties of the dwarfs when they are studied in isolation, rather than as part of a full application with many interacting parts.

The first application is a sorting benchmark called MilliSort. Sorting has been used to evaluate system performance for many decades, originating with a challenge proposed by Jim Gray and others in 1985 [2]. Sorting plays an important role in many distributed computations. For example, sorting can be used as a data preprocessing step to support efficient range queries, to improve data locality for graph partitioning [66], and to perform load balancing [66, 21, 39]. Sorting is also very challenging because it requires intensive and unpredictable communication (any record can potentially end up on any server). This creates challenges both for the algorithm and the underlying infrastructure.

For MilliSort, the goal is to sort as many small records as possible in intervals around one millisecond, using any number of servers in a datacenter. Each record contains 100 bytes, consisting of a 10-byte key and a 90-byte value. Before starting the benchmark, the MilliSort application is pre-loaded and the unsorted records are distributed evenly among the available machines in DRAM. The data on each server is structured with all of the keys in a single block of memory and all the values in another block, in corresponding order. Upon completion, the data must be redistributed across the same servers, sorted such that one server contains the records with the smallest keys, and so on. At the end of the sort, the data on each server is structured in two blocks of memory, one containing the keys in sorted order and another containing the values in the same order as their keys. The challenge does not require that the result data be distributed evenly across the servers, but this turns out to be essential for good performance.

The second "application" is a collection of three SQL queries from the documentation of Google's BigQuery [95, 56]; these queries are referred to as MilliQuery collectively. I expect that many flash burst applications will perform data analytics to provide real-time results from ad hoc queries; the goal for MilliQuery is to capture dwarfs that will be common in these applications.

These three queries are chosen with different levels of complexity, in order to span a range of SQL behaviors (see Listing 1).

Q1: counts the number of article views on Wikipedia by language (there are at most a few hundred

```
/* MilliQuery Q1: count article views on Wikipedia by language */
SELECT language, SUM(views)
FROM `bigquery-samples.wikipedia_benchmark.Wiki1B`
GROUP BY language

/* MilliQuery Q2: top 10 IPs by the number of edits to Wikipedia */
SELECT contributor_ip AS ip, COUNT(*) AS count
FROM `publicdata.samples.wikipedia`
GROUP BY ip ORDER BY count DESC LIMIT 10

/* MilliQuery Q3: complex data analytics on GitHub data */
WITH
  repo_authors AS (    -- Build the intermediate author table
    SELECT repo_name, author.name AS author
    FROM `bigquery-public-data.github_repos.commits`,
      UNNEST(repo_name) AS repo_name
    GROUP BY repo_name, author),
  repo_languages AS (  -- Build the intermediate language table
    SELECT lang.name AS lang, lang.bytes AS lang_bytes, repo_name
    FROM `bigquery-public-data.github_repos.languages`,
      UNNEST(language) AS lang)
SELECT lang, author, SUM(lang_bytes) AS total_bytes
FROM (repo_languages JOIN repo_authors USING repo_name)
GROUP BY lang, author ORDER BY total_bytes DESC LIMIT 100
```

Listing 1: The three SQL queries used in the MilliQuery benchmark

	Coordination	**Shuffle(s)**	**Dwarf(s)**
MilliSort	Heavy	≥2	Sort
MilliQuery Q1	None	0	Aggregate
MilliQuery Q2	Light	1	Repartition,Aggregate
MilliQuery Q3	Light	3	Repartition,Join

Table 2.1: A comparison of the applications used for studying flash bursts.

different languages).

Q2: finds the top 10 IP addresses by the number of edits they made to Wikipedia articles.

Q3: for each combination of author and programming language, sum all of the bytes in that language in any repository for which the author was a contributor; returns the top 100 author-language pairs.

In each case, the goal is to process as much data as possible within 1 ms using unlimited datacenter resources, assuming that the application is pre-loaded and data is initially distributed uniformly across the nodes.

Table 2.1 compares these applications in terms of the complexity of coordination (how difficult it is to divide the work among the participating nodes and coordinate their behaviors), the number of shuffles required, and the dwarfs captured. Together, MilliSort and MilliQuery cover a significant range of interesting behaviors.

The goal for MilliSort and MilliQuery was to push the idea of flash bursts to the extreme. It is

hard to know what burst sizes will prove useful to applications, so I set out to characterize the range that is practical: what is the smallest time interval and what is the largest number of servers that can be harnessed efficiently? I also hoped to learn what are the technical factors that limit flash bursts. I chose a one millisecond time limit because it seemed like an extreme goal: at the outset, I was unsure whether it would be possible to do anything useful in such a short interval.

2.3 The MilliSort algorithm

Although distributed sorting is not new, designing a sorting algorithm to operate at a timescale of one millisecond introduces new challenges due to the high cost of coordination. This section describes MilliSort's algorithm in detail and presents a novel hierarchical approach to data partitioning that enables efficient coordination even at very small timescales.

Most distributed sorting algorithms use a partitioning approach, and MilliSort follows this tradition. First, the data is partitioned by deciding which key ranges should end up on each server; then the records are shuffled between servers to implement the chosen partitions. This approach optimizes the use of network bandwidth by transmitting each record only once, during the shuffle phase. The partitioning approach makes sense because it optimizes the use of network bandwidth, which has traditionally been the scarcest resource in distributed sorting. Alternative approaches, such as those that use multi-stage merge sorts, require data to be transmitted over the network multiple times, so they have proven slower than the partitioning approach.

More precisely, MilliSort implements a variant of distributed bucket sort, with one bucket for each server. It consists of four phases:

- Local sort: each server sorts its initial data.
- Partition bucket boundaries: the servers collectively determine the key ranges that will end up on each server after sorting.
- Shuffle: each server transmits its keys and values to the appropriate targets.
- Local rearrangement: the data arriving on each server during the shuffle phase must be rearranged into two totally sorted arrays, one for keys and one for values.

The sections below discuss each of these phases in more detail. The partitioning phase is presented first, since it is the most complex phase and also the most interesting phase from an algorithmic standpoint.

2.3.1 Histogram sort

One of the most widely used partition-based sorting algorithms is histogram sort, which computes the final key ranges by iteratively refining an existing partition until the keys are evenly distributed

on the servers. A typical workflow of histogram sort is as follows. In the beginning, a central server picks $M-1$ splitters, which divide the key space into M buckets, and broadcasts them to the other servers. Then, each server computes a local histogram of its keys in the buckets and sends it back to the central server. Finally, the central server computes a global histogram by summing up the local histograms and adjusts the splitters to reduce the imbalance. The process of histogramming and refinement is repeated until an even partition is achieved. In addition to the one mentioned above, there are other variations of histogram sort which use more splitters for histogramming or increase the number of splitters as they progress.

However, histogram sort is undesirable for MilliSort since it requires many iterations to converge, and each iteration incurs significant message delays. To avoid overloading the central server, histogram sort often uses group communication to broadcast splitters and reduce local histograms in a tree structure. As a result, each iteration incurs $2log(M)$ back-to-back message delays. In the test cluster, the combined cost of broadcast and gather is at least 50 μs for 100 servers. With just 10 iterations, message delays alone will take away half of the 1 ms time budget. Finally, the actual cost of partitioning will be even higher due to other overheads; the reported partitioning times of some recent histogram sort implementations easily exceed 50 ms for 512 HPC nodes [53, 37].

2.3.2 Sample sort partitioning

MilliSort takes a different approach to partitioning, selecting a larger number of initial keys so that it can estimate the distribution in a single iteration. MilliSort's partitioning algorithm is based on sample sort with regular sampling [87, 59]; the basic idea is to select many keys from the starting data, use these to estimate the distribution of keys, and pick partition boundaries based on the estimated distribution. Figure 2.2 shows the basic idea. After sorting its local data, each server samples its keys at equally-spaced intervals within the sorted records; these samples are referred to as *pivots*. The pivots of all servers are collected and sorted (more details on this below). Finally *splitter* values are chosen from the sorted pivots. If there are M machines participating in the sort, $M-1$ splitters are chosen, which divide the sorted pivots into M equal-size groups. The splitters determine how records are divided between servers during the shuffle phase: server i will eventually hold all records with keys greater than or equal to the ith splitter and less than the $i+1$th splitter.

Because of the sampling used by this approach, there is no guarantee that each server will end up with exactly the same number of records at the end of the sort. If there are N total records divided among M machines and each machine chooses sM pivots, then, in the worst case, a server may end up with as many as $(1+1/s)(N/M)$ records (N/M is the ideal number in a perfect partition) [87, 59]. For MilliSort, s is set to 1 (each machine chooses M pivots), so the final partition sizes are guaranteed to be within a factor of 2 of the ideal size. In practice the distribution of records is considerably more uniform than suggested by the worst-case formula above.

With this approach, the total number of pivots to sort is sM^2. This means that as the number

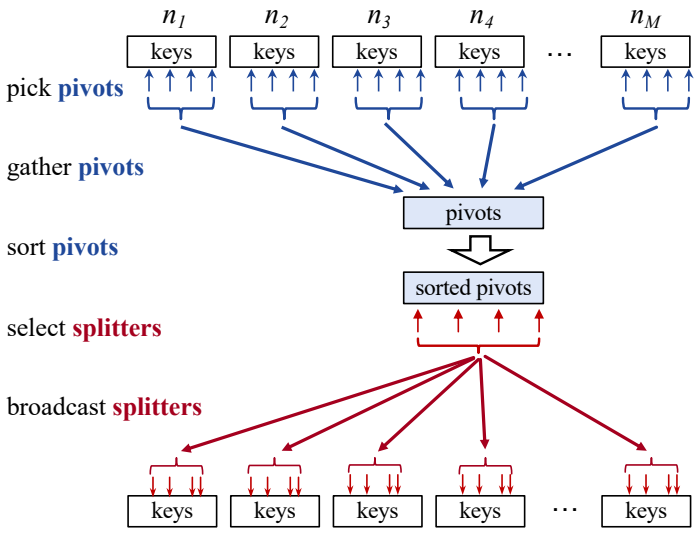


Figure 2.2: A basic sample sort partition mechanism; n_1 – n_M are the MilliSort nodes.

of machines increases, partitioning will take more and more time, even if all of the machines share the work. Given a limited amount of time for the sort, partitioning cost will limit the number of machines that can be harnessed.

2.3.3 Recursive partitioning

One way to perform the partitioning is to gather all of the pivots on a single coordinator server, sort them locally on that server, then broadcast the splitters back to all of the servers. However, this approach is too inefficient for MilliSort. If 300 machines participate in the sort, there will be 90,000 pivots; as shown in Figure 2.1, a single server can only sort about 40,000 keys in one millisecond, so the sorting alone would take more than 2 ms. The overhead of receiving all the pivots on a single server is also problematic. Thus, millisecond-scale sorting requires partitioning to be performed in a distributed fashion.

MilliSort uses a recursive approach to partitioning: the pivots are sorted in a distributed fashion using a smaller instance of MilliSort, as shown in Figure 2.3. A subset of the machines, called *pivot sorters*, sort the pivots and select splitters; each of the other servers is assigned to one of the pivot sorters. To begin the sort, each pivot sorter gathers the pivots from its assignees. The pivots arriving from each source machine are already sorted, so the pivot sorter can use merge sort on the arriving data to produce a sorted list of all the pivots for which it has responsibility. Then each pivot sorter samples its pivots to choose a smaller number of *level 2 pivots*; the level 2 pivots are passed to a coordinator, which sorts them and produces a set of *level 2 splitters*. The coordinator broadcasts the level 2 splitters back to the pivot sorters, which then perform a shuffle to redistribute the pivots among the pivot sorters in sorted order.

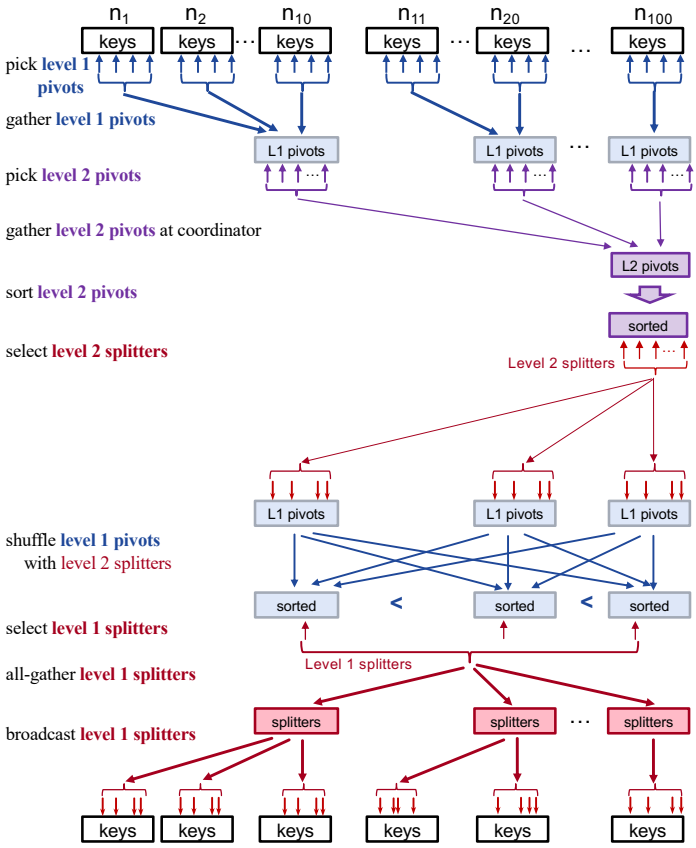


Figure 2.3: A two-level partitioning example with $M = 100$, $s = 1$, $r = 10$.

At this point the pivots have been sorted and splitters must be chosen (i.e., every sM^{th} pivot across all of the pivot sorters must be selected). It would be nice for each pivot sorter to independently select splitters from its pivots, but in order to do this, the pivot sorter must know its rank (i.e., how many pivots stored on other servers are smaller than the pivots that it stores). The rank is not immediately obvious because pivots are not distributed uniformly across the pivot sorters. The solution is to distribute rank information during the shuffle phase of the pivot sort. When a pivot sorter sends a group of pivots to another pivot sorter during the shuffle, it includes the *local rank* of that group (i.e., the number of pre-shuffle pivots from that pivot sorter that are smaller than those in the group). Each pivot sorter can determine its rank by summing the local ranks in all of the shuffle messages it receives. Once a pivot sorter knows its rank, it can identify the splitters that it stores.

Finally, once the splitters have been determined, they must be disseminated to all of the machines participating in the sort. One approach would be for each of the pivot sorters to broadcast its splitters to all of the M machines; however, this would have a high cost because of the large number

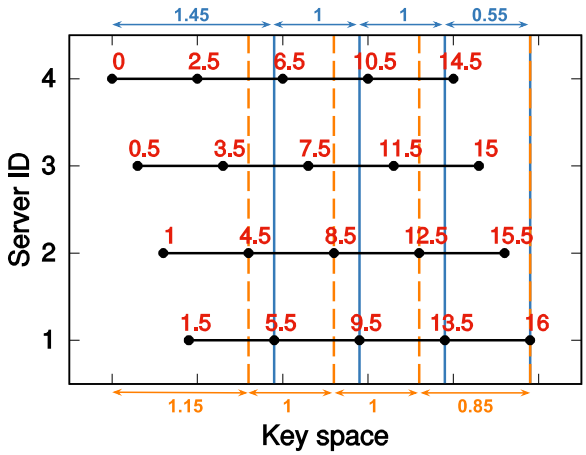


Figure 2.4: A comparison of the splitter selection algorithms on an example scenario. The keys of each server's data are uniformly distributed in a range indicated by a horizontal line, where dots are pivots (starting and ending pivots are not considered by the original algorithm). The splitters selected by the original algorithm are indicated with solid blue vertical lines. The splitters selected by the weighted algorithm are indicated with dashed vertical lines. The numbers on pivots are the cumulative pivot weights used by the weighted algorithm. The numbers on arrows above and below the diagram indicate the relative sizes of the buckets for the original and weighted algorithms, respectively.

of messages that would result. Instead, MilliSort uses a two-step approach to distribute the splitters. In the first step, an all-gather operation is used to exchange the splitters among the pivot sorters, so that each pivot sorter has all $M-1$ splitters. Then each pivot sorter broadcasts the complete set of pivots to all of the machines assigned to it.

If the number of servers is very large, the 2-level approach described above will still take too long. If that is the case, additional levels may be used in the partition. For example, in a 3-level approach the level 2 pivots will not be sorted on a single coordinator; instead, they will be collected by a smaller number of second-level pivot sorters, which will then select a set of level 3 pivots. The level 3 pivots will be collected and sorted on a single coordinator, resulting in level 3 splitters, which are used to shuffle the level 2 pivots. The approach can be extended to an arbitrary number of levels, though the experiments suggest that 2 or 3 levels is appropriate for MilliSort.

Let r be the reduction factor at each level of recursive sorting. For M total machines, there will be M/r pivot sorters, M/r^2 second-level pivot sorters, and so on. For the MilliSort implementation a reduction factor of 10 is used.

2.3.4 Improved splitter selection

The splitter selection algorithm described in Section 2.3.2 tends to produce unbalanced partitions where the first server contains considerably more records than the last server. This imbalance can

	120 servers (7000 records per server)							
	Uniform				Gaussian			
	Naive		Improved		Naive		Improved	
Pivots	P50	P90	P50	P90	P50	P90	P50	P90
120	33.6%	45.5%	6.9%	8.6%	37.6%	47.7%	6.9%	8.6%
240	23.5%	25.4%	4.4%	5.4%	23.4%	25.4%	4.4%	5.5%
360	15.9%	16.9%	3.1%	3.8%	15.9%	17.0%	3.2%	3.9%
480	11.5%	12.3%	2.5%	3.0%	11.6%	12.4%	2.5%	3.0%
600	9.6%	10.3%	2.0%	2.4%	9.7%	10.4%	2.0%	2.5%

Table 2.2: Excess records in the largest partition, relative to the average partition size, when using the improved splitter selection algorithm, vs. the naive algorithm. Input keys were drawn from two random distributions, and rows corresopnd to different numbers of pivots per server. "P50" and "P90" represent the median and 90th percentile over 1000 runs, respectively.

be explained by considering the groups of keys delimited by the pivots from each server; all of the groups contain about the same number of records. If we choose the Mth smallest pivot as the first splitter (assuming $s = 1$), then the first server will contain M full groups of records. In addition, it will contain some records from up to $M - 1$ additional groups, whose contents are divided between the first two servers (see Figure 2.4). Thus, the first server is likely to contain about $1.5M$ groups worth of records. In contrast, the last server will contain records from at most M groups, of which all but one are partial (some of their records have keys smaller than the last splitter). Thus, the last server will probably hold only about $0.5M$ groups of data.

To mitigate the data imbalance, I developed a new weighted approach to selecting splitters. The original approach behaved as if all of the keys in each group had the same value as the pivot at the end of the group. The new approach changes the weighting, so that half of the keys in each group are attributed to the beginning of the group and half to the end. Specifically, in the new approach, each server also includes its smallest and largest keys as pivots, and each pivot is annotated with a weight. The first and last pivots have a weight of 0.5 (half a group), and middle keys have a weight of 1.0 (half of the preceding group and half of the following group). The annotated pivots from all servers are collected and sorted as usual. Then the pivots are scanned from smallest to largest, adding up the pivot weights; when a given pivot is reached, the cumulative weight is an estimate of how many groups worth of data have keys less than or equal to the pivot. The ith splitter will be the first pivot encountered where the cumulative weight is at least $i \times sM$.

I evaluated the improved mechanism for splitter selection by running stand-alone simulations with keys drawn from random distributions. Table 2.2 shows that the improved mechanism ensures that the largest partition is within 10% of the ideal size. Without the improvement, the largest partition will be 30-50% larger than ideal if the number of pivots per server equals the number of servers. Said another way, using the new splitter selection mechanism provides a greater benefit than increasing the pivots per server by 5x.

2.3.5 Local sort

Now that the partitioning mechanism has been described, the next few subsections discuss the remaining phases of MilliSort. These phases are more straightforward than the partitioning phase, though their performance is still important.

MilliSort relies on prior work for the local sort. The problem of sorting on a single compute server is well-studied, with many solutions that can take advantage of multiple cores. In the experiments the In-place Parallel Super Scalar Samplesort (IPS^{4}o) algorithm [7] is chosen as a reasonable representation of a multi-core comparison-based sorting algorithm.

Once the keys have been sorted, the values must be rearranged to match the order of the keys. This can be overlapped with the partitioning phase and the key shuffle; the rearranged values are not needed until the value shuffle.

2.3.6 Shuffle

Once the splitters have been selected and distributed to all of the machines, MilliSort uses an all-to-all shuffle to transmit each key and value to the appropriate server. The keys and values on each server were already sorted during the local sort phase. Each server uses the splitters to determine the range of keys to send to each other server. It then sends a message to that server containing the appropriate range of keys, followed by the data associated with those keys.

2.3.7 Local rearrangement

Within each message that a server receives during the shuffle phase, the keys and values are in sorted order. The server must then combine these chunks into two sorted arrays, one containing all the keys and other containing all the values. To do this, each server first performs a merge sort on the arrays of keys; then it rearranges the values to match the order of the keys. Each key contains its index in the incoming shuffle message (which is the same as the index of the value in the message). Once the keys have been sorted, a sequential scan is made over the key array; the index stored with each key is used to find the corresponding value and the value is then stored at the next sequential location in the result array. This two-step approach is faster than a one-step approach that merges both keys and values together, because it copies the (larger) values only once.

2.4 Implementation

2.4.1 Group communication

I created a C++ library that implements the group communication primitives described in Section 2.1.4; both MilliSort and MilliQuery make extensive use of this library. The group communication primitives are implemented using the network transport infrastructure from RAMCloud [77]. RAMCloud's transports use kernel bypass with either DPDK [20] or the Infiniband verbs interface to provide low latency (5 μs round-trips) and high throughput (up to 25 Gbps). However, RAMCloud requires all network communication to pass through a single dispatcher thread, which limits message throughput to about 1.6 million messages per second. The group communication primitives contain 1500 lines of C++ code, not counting code in RAMCloud.

Broadcast, gather, and all-gather were implemented using well-known approaches [93, 97]. For broadcast, MilliSort uses a k-nomial tree, with the topology optimized based on precise knowledge of message latency and per-message cost. For gather, MilliSort uses a k-nomial tree with $k = 6$, in order to utilize as much network bandwidth as possible at each step. For all-gather, MilliSort uses a hypercube approach, extended to handle group sizes that are not even powers of two [85].

Shuffle is the most important of the group communication primitives: it accounts for half or more of the end-to-end time in the highest performing MilliSort configurations and it is also used in two of the three MilliQuery queries. However, achieving high-performance for shuffle is challenging. Ideally, shuffles should utilize the full bandwidth of the network, with each host simultaneously sending and receiving at the speed of its uplink. However, achieving this goal is difficult. One problem is small messages, which are more likely to occur in flash bursts than other applications (see below). Another problem is that shuffle requires full bisection bandwidth in the underlying network. Fortunately, the test cluster has full bisection bandwidth. Unfortunately, I was unable to harness all of the available bandwidth.

Achieving full bisection bandwidth requires a near-perfect bipartite matching, where at any given time each source transmits at full bandwidth to a different target. This is difficult to achieve, for two reasons. First, two servers might attempt to transmit simultaneously to the same target. When this occurs, bandwidth is wasted on the senders, since the target can only receive from one of them at a time; in addition, some other server will not be receiving anything at all, which wastes its incoming bandwidth. Second, achieving full bisection bandwidth requires perfect load-balancing across the network fabric. Unfortunately, the test cluster does not support packet-level load-balancing. Instead, it uses flow-consistent hashing, where all packets from a particular source to a particular destination are routed over a single (randomly chosen) path through the fabric. Even if two sources send to different targets, their routes might traverse a common intermediate link, resulting in bandwidth underutilization. With large clusters, routing conflicts are virtually guaranteed.

Initially, shuffles were implemented by handing off the messages to the underlying network transport of RAMCloud; by default, RAMCloud uses a protocol called Homa [69] that schedules messages based on the SRPT (shortest remaining processing time first) policy. However, after observing poor performance of this naive implementation, I attempted a lock-step approach to construct a bipartite matching. In the lock-step approach, in step i each server n transmits to server $(n+i)$ mod M, and the start of step $i+1$ is delayed until step i has completed. This mostly eliminated the problem where two senders transmit to the same target, but it works best when all messages are large and the same size. If messages are small, or if messages have different sizes, the act of maintaining lock-step wastes most of the network bandwidth (e.g. each step must wait for the longest message in the preceding step to complete). Furthermore, lock-step is still vulnerable to conflicts in network routes. As a result, the performance with this approach was far from satisfactory.

I then switched to a nearly-opposite approach, implementing shuffles in a "high-entropy" fashion that is granular, concurrent, and random. The first step is to ensure that messages sent during shuffles are relatively short (at most 40 KB in the current implementation). In many cases, the messages are inherently short; if the data from one server to another exceeds a threshold size, it is divided into multiple short chunks, which are sent as separate messages. Each host sends multiple messages concurrently; the targets are chosen randomly, but a given source will have at most one message outstanding to a given target at a time. With this approach there will still be conflicts but they will not stall senders; conflicts simply result in packet queueing in the network. Since each sender has multiple outstanding messages, it is likely that there will be incoming data for each server at all times. Since messages are short, buffer overflows in the network are unlikely and a sender doesn't waste much time on a busy receiver before directing its bandwidth elsewhere. Stalls will occur only at the end of the shuffle, when a sender is waiting for its last few messages to complete. The high-entropy approach resulted in much better performance than the alternatives I tried before it[1].

Shuffles are more challenging in a flash burst than in more traditional environments that operate at large timescales. A flash burst scales by increasing the number of servers, with less data on each server, in order to complete more quickly. Less data on each server means the size of each shuffle message will drop; more servers means that each server's data is split among more shuffle messages, which also results in less data per message. The result is a rapid drop in shuffle message size as a flash burst scales. This leads to low network bandwidth utilization and high per-message overheads.

One possible solution is to use a two-level shuffle to reduce the number of messages sent and received by each server from $M-1$ to $2\cdot(\sqrt{M}-1)$. Two-level shuffle arranges all servers into a virtual mesh and proceeds in two rounds: each server first exchanges data with other servers in the same row, and then in the same column. However, this approach doubles the network bandwidth consumed, since most data must be transmitted twice. For the experiments the increased bandwidth usage of

[1] This high-entropy approach was developed during the MilliSort/MilliQuery project as a workaround to improve the shuffle performance; it is not the new shuffle algorithm that will be described later in the dissertation.

a two-level shuffle was more problematic then per-message overheads for a single-layer shuffle; I did not find any situations where multi-level shuffles are advantageous.

2.4.2 MilliSort

The MilliSort implementation is fully decentralized: each server operates independently in an event-driven style, with no central coordinator (central coordination is intolerable for flash bursts, both because of the latency it adds and also because central coordination often involves lock-step operation at the end of each stage, which suffers from stragglers). While the order of stages executed in a server is well defined, the timing is affected by remote procedure calls (RPCs) from other servers. At various points, the progress of the server will stall until certain pieces of data have arrived. As one example, a pivot sorter cannot select level 2 pivots until it has received pivots from all of the servers in its group.

Different servers must have different behaviors during the sort. For example, only a subset of the servers will act as pivot sorters. Each server has an identifier ranging from 0 to $M - 1$, which determines the various roles it will serve during the sort. For example, servers with identifiers that are 0 mod r serve as pivot sorters and they receive pivots from servers with the following $r - 1$ identifiers. At the start of the sort each server knows its identifier, the value of M, and the addresses of the other servers.

Achieving the highest overall performance for MilliSort requires careful optimization of both communication and computation. Section 2.4.1 has already discussed the challenges associated with shuffles. Using cores efficiently is another challenge, because the number of threads changes rapidly over the life of the sort. Thus, MilliSort uses Arachne [83] for efficient user-level thread and core management. For example, Arachne allowed us to quickly spawn a group of threads to parallelize a task, such as local sort, and place them precisely on all available cores. In addition, MilliSort creates a new thread for each incoming RPC and leaves it to Arachne to schedule those threads on cores that are less busy.

2.4.3 MilliQuery

I created a special-purpose implementation of each of the three MilliQuery queries, largely based on the query plans generated by BigQuery. These queries were much easier to implement than MilliSort, particularly given the availability of the group communication library. The implementation took only a few days, and the three queries contain 250, 300, and 800 lines of C++ code, respectively.

The implementation of Q1 follows a simple scan-aggregate pattern: each server scans its data independently to count the views by language, then the local results are gathered back to one server, using a k-nomial tree and combining the statistics at each node. Since the number of distinct languages is only a few hundreds, all messages in the gather phase are small.

Similar to Q1, Q2 can also be implemented as local scan followed by a gather. But, the number of distinct IP addresses is quite large, so the gather phase would consume too much network bandwidth with this naive approach. Thus, before the gather phase, a shuffle is used to collect all the counts for each address in one place, using a hash partition. Then each node in the gather tree collects local results from all its children, but only needs to send the top ten IP addresses to its parent.

Q3 is considerably more complex and requires a total of three shuffles. First, two shuffles are used to materialize two intermediate tables, distributing the records for each table using a hash partition with the `repo_name` field. Then, the two tables are joined locally using the same key, and a third shuffle redistributes the joined records by hash partitioning on `(lang, author)`. Finally, top-100 statistics are computed locally and aggregated as in Q2.

For simplicity, I didn't implement additional mechanisms to handle hot keys in hash partitioning for Q2 or Q3; however, it's possible to use MilliSort's recursive partitioning scheme to create a more balanced range partitioning, at the expense of higher partitioning cost.

2.4.4 Communication infrastructure

I chose to build the MilliSort and MilliQuery prototypes on top of RAMCloud [77] mainly to reuse its flexible network infrastructure. However, this choice comes with the cost of limited message throughput for two reasons:

- The single dispatch thread in RAMCloud has relatively high per-message costs (all messages must pass through the dispatch thread, which results in expensive cross-core communication) and presents a central bottleneck which prevents us from achieving higher throughput by adding more cores.
- The underlying Omni-Path PSM2 library [43] that is responsible for sending and receiving packets is actually not a packet I/O driver but a reliable message-passing transport with its own congestion/flow control, which adds considerable overhead; in addition, the Omni-Path host fabric interface (HFI) does not have a lightweight mechanism for transfering small chunks of data via DMA, so packets are sent and received using programmed I/O, which leads to higher CPU overhead and, even worse, cache pollution.

As a result, a single server cannot fully utilize its network bandwidth when the messages are relatively small. In the experiments, I decided to place multiple servers on each machine, each with its own dispatch thread, to increase the network utilization. This approach can scale the overall message throughput of each machine linearly, but it also incurs higher coordination overhead due to the increased number of servers. Future implementations that are based on a more efficient networking stack such as [50, 28] could eliminate the need to run multiple servers per machine.

CPU	Xeon Gold 6148 (2 sockets × 20 cores @ 2.40GHz)
RAM	384 GB DDR4-2666
Networking	100Gbps Intel Omni-Path Interconnect

Table 2.3: The hardware configuration used for benchmarks. All machines ran CentOS 7.3.1611 (Core) with hyperthreading disabled. The network fabric used a two-level fat-tree topology to provide full bisection bandwidth at 100 Gbps per machine.

	Total records processed (# servers used)			
Time budget	**MilliSort**	**Q1**	**Q2**	**Q3**
1 ms	0.84 M (120)	47.6 M (280)	6.72 M (140)	0.034 M (60)
10 ms	26 M (280)	980 M (280)	224 M (280)	2.24 M (280)
Scaling	31.0x (2.3x)	20.6x (1x)	33.3x (2x)	67.9x (4.7x)

Table 2.4: Overall performance of MilliSort and MilliQuery.

2.5 Performance measurements

The goal in evaluating MilliSort and MilliQuery was to answer the following questions:

- How much data can be processed in one millisecond (or ten milliseconds), and how many servers can be harnessed efficiently in each interval?
- How does application behavior change if the time budget is increased or decreased?
- What factors limit the applications' performance and scalability, and what stresses do these applications create for underlying infrastructure?
- How effective is MilliSort's new partitioning mechanism?
- How efficient is MilliSort compared to purely local sort or other distributed sorts?

The hardware configuration used in the evaluation is shown in Table 2.3. To better utilize the network bandwidth (discussed in Section 2.4.4), four independent servers were run on each machine, two on each socket. During the experiments, a total of 70 machines were available in the cluster, which allowed up to 280 servers; and there were no competing tasks running on the machines. All MilliSort experiments used 2-level partitioning.

2.5.1 Overall performance

I varied the amount of data per server and the size of the cluster to find the largest amount of data that can be processed by each of the applications in either 1 ms or 10 ms; Table 2.4 shows the results, along with the best configurations for each interval. With a 10 ms time budget all of the applications can harness all 280 available servers and they can process 2.2–224M records, depending on the application. A 1 ms time budget is more challenging for most of the applications. Only

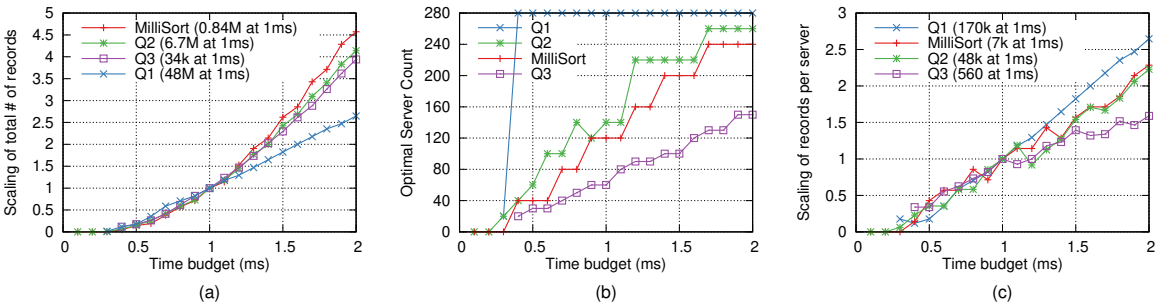


Figure 2.5: Scaling properties of MilliSort and MilliQuery as a function of time budget: (a) size of dataset processed (normalized to a value of 1.0 for a 1 ms time budget for each benchmark); (b) cluster size that yields the largest dataset processed; (c) records per sever at largest dataset processed (also normalized to a value of 1.0 for a 1 ms time budget).

MilliQuery Q1 can use all of the servers; the other applications ranged from 60–140 servers. The amount of data processed in 1 ms varied dramatically among the applications, from a low of 34K records in MilliQuery Q3 to a high of 48M records in Q1.

2.5.2 Quadratic scaling

The total number of records that can be processed in an interval varies quadratically with the size of the interval. If the time budget increases by a factor of X, it can be observed that both the data handled by each server and the number of servers increase by at least X, for a total increase in throughput of at least X^2. This effect can be seen in Figure 2.5. Figure 2.5(a) shows that all of the benchmarks except Q1 exhibit quadratic or better scaling as the time budget increases from 1 ms to 2 ms. Q1 would scale exponentially if more servers were available, but it already used all of the available servers at 1 ms, so it can only scale linearly by increasing the amount of data per server. Figure 2.5(b) shows optimal cluster size as a function of time budget. All except Q1 exhibit almost linear scaling. Figure 2.5(c) shows the scaling of the per-server dataset size as the time budget increases. All except Q3 scale at least linearly, after a fixed initial overhead. Q3 scales per-server records slightly less than linearly.

The quadratic behavior can also be seen for Q3 in Table 2.4: its throughput scales by 68x as the time budget increases from 1–10 ms. Scaling for the other applications becomes limited by the available servers long before reaching the 10 ms time budget; Figure 2.5(b) shows that MilliSort, Q1, and Q2 have consumed almost all the available servers with a 2 ms time budget. Linear scaling of servers suggests that these applications could harness at least 1200 servers with a 10 ms budget (5x the number at 2 ms).

	120 servers (0.84M records)				**240 servers** (1.68M records)			
Phase	Mean	Max	%	Max/Min	Mean	Max	%	Max/Min
Local Sort	147.0	216.3	22.2%	1.83	137.8	202.3	12.7%	1.77
Partitioning	200.5	214.4	22.0%	1.16	410.4	428.6	26.9%	1.11
Pivot Shuffle	83.2	87.9	9.0%	1.13	219.3	240.1	15.1%	1.27
Shuffle	377.2	402.8	41.3%	1.16	738.9	789.0	49.6%	1.14
Rearrangement	128.1	142.7	14.6%	1.18	146.9	173.3	10.9%	1.26
Total	942.3	976.0	100%	1.09	1523.8	1591.1	100%	1.08

Table 2.5: Time breakdown of each MilliSort phase with two different cluster sizes, where per-node dataset sizes are fixed at 7000 records. All times are in μs and reflect the median over 200 runs. "%" is the time spent by the slowest server for that phase, as a fraction of total time. "Max/Min" is the ratio of times for the slowest and fastest servers for that phase. "Partitioning" includes the time spent on "Pivot Shuffle".

2.5.3 Scaling below 1 ms

Quadratic scaling also means that throughput drops rapidly with time budgets less than 1 ms. This is visible in Figure 2.5(a). With a time budget of 0.5 ms, throughput has dropped by more than 4x for all of the applications, and none of the applications has appreciable throughput for budgets less than 0.5 ms. For these applications, the lower bound on useful timescale is around 0.5–1.0 ms with the current per-message overheads.

2.5.4 Limiting factors for scalability

There are two primary factors that limit the ability of the applications to scale up in servers or down in time: coordination and shuffles. The costs of both activities increase with the cluster size; when the amount of data per server is zero, these are essentially the basic costs of harnessing servers. Table 2.5 illustrates the effect of these two factors by showing the cost of each MilliSort phase for two different cluster sizes with the records per server held constant. In the 120-server configuration, partitioning and shuffles take 63% of the total running time. If the cluster size is doubled, then the time taken by these two activities is also doubled (1218 μs vs. 617 μs), even though each server still processes roughly the same amount of data. The fraction of total running time consumed by coordination and shuffles increases from 63% to 77%. At the same time, the combined cost of other phases remains almost constant in both configurations. In general, when the time budget is held constant, larger clusters result in larger basic costs, so less time is left for actual work such as local computation and data transfer. As a result, this limits the ability to harness more servers within the time budget or perform meaningful flash bursts in smaller timescales.

Figure 2.6 and 2.7 provide more details of the two limiting factors for MilliSort and MilliQuery by showing how the total processing time changes with the cluster size and amount of data per server. In most graphs of Figure 2.6 and 2.7, the lines for different cluster sizes are roughly parallel, indicating that the marginal cost of handling additional data is about the same for all sizes (the

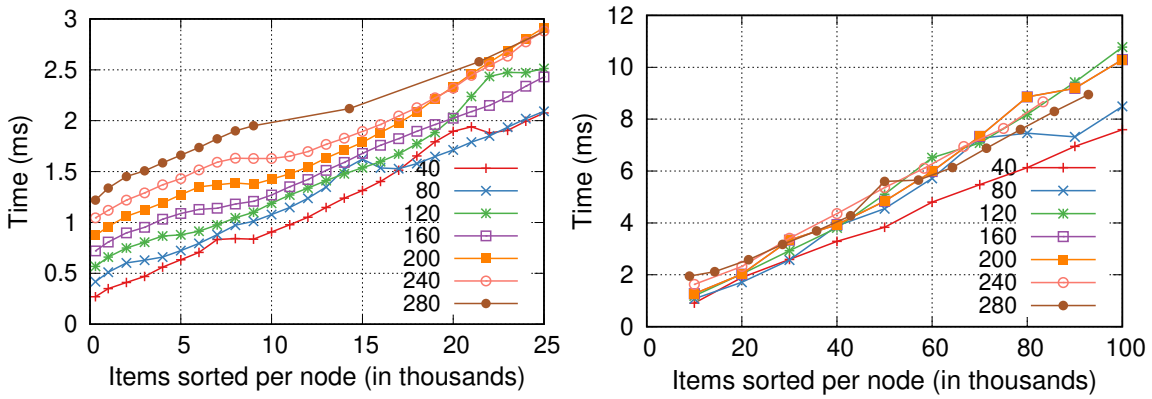


Figure 2.6: Total sorting time for MilliSort as a function of records per server, with different cluster sizes. The left figure assumes a 1 ms goal and the right figure assumes 10 ms. Each line represents a different cluster size (the number in the legend is the server count). Each data point is the median time from 200 runs.

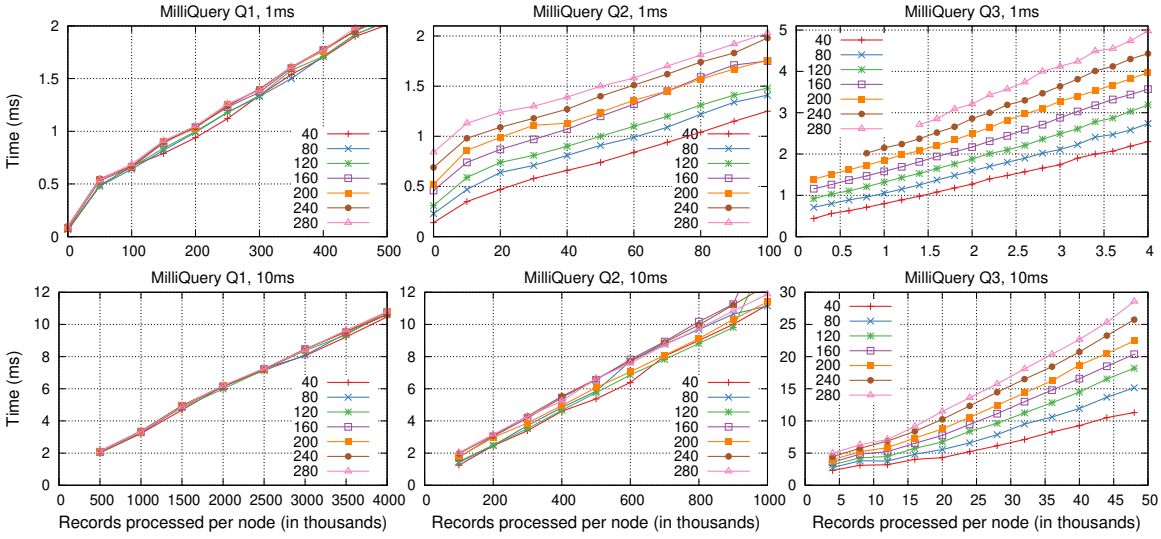


Figure 2.7: Total processing time for the three MilliQuery queries as a function of records per server, with different cluster sizes. From left to right, the figures represent Q1, Q2, and Q3 respectively. Each data point is the median time from 100 runs.

marginal cost starts higher when the amount of data per server is smaller, but plateaus out quickly once the benefit of batching diminishes). However, larger cluster sizes have larger fixed overheads (y-intercepts of the lines), which consist of the partitioning cost plus the per-message costs of the shuffles (a shuffle must send one message to each peer, even if it only contains a single record). These figures also show the diminishing returns at timescales less than 1 ms: even as the number of records per server approaches zero, running times remain 0.3–1.2 ms and 0.15–0.84 ms for MilliSort and MilliQuery Q2 respectively, depending on cluster size.

MilliQuery Q3 difers from the other applications in that its total processing time increases more than linearly with the input records per server, and the rate of increase is higher for larger clusters (the gaps between the lines in the right graphs of Figure 2.7 become wider for larger numbers of input records). This is because Q3 uses join operations where the number of output records tends to increase quadratically with the number of input records, so the number of input records doesn't reflect the actual number of records each server has to process.

Different applications have very different fixed overheads of harnessing servers. Q1 has the smallest fixed overheads and they are about the same for all cluster sizes. This is because Q1 requires very little coordination and doesn't use shuffle; the only communication primitives used by Q1 are broadcast and gather, whose overheads increase only in log scale with the cluster size. MilliSort and Q2 are very similar: their fixed overheads increase almost linearly with the cluster size since both applications require a big shuffle at the end. However, for the same cluster size, the fixed overhead of MilliSort is higher due to the additional partitioning cost (with 2-level partitioning, this cost also increases almost linearly with the cluster size; this will be discussed in Section 2.5.6). For the same reason, the lines of Q2 in the graphs are closer than the lines of MilliSort. Finally, the fixed overheads of Q3 are about 3x higher than Q2 for all cluster sizes (unfortunately, it's hard to see in the graphs), as Q3 requires a total of three shuffles.

Figures 2.6 and 2.7 also show that efficiency improves with increasing records per server. For MilliSort, once the number of records per server reaches about 25000, the coordination costs become small compared to other factors and the shuffle efficiency improves, so there is little difference in overhead between different cluster sizes (the 40-server cluster remains significantly more efficient than the other cluster sizes because it avoids contention in the network core; this will be discussed in Section 2.5.5). The closeness of the lines in these 10 ms graphs indicates that all applications except Q3 could easily harness many more than 280 servers efficiently with a 10 ms time budget.

The next subsections discuss shuffle and partitioning costs in more detail.

2.5.5 Shuffle efficiency

Shuffles present the most significant challenge to scalability in the experiments, especially at 1 ms time scales. This is reflected in Table 2.4: Q1, which has no shuffles, can harness far more servers and process far more data than the other benchmarks, while Q3, which has three shuffles, is the

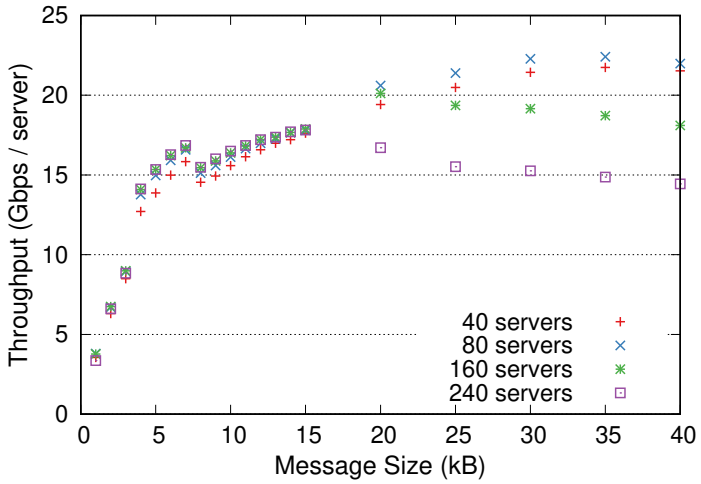


Figure 2.8: Stand-alone shuffle benchmark performance as a function of the cluster size and the message size (amount of data sent from each source to each destination).

least scalable. MilliSort and Q2, which each use one shuffle, fall in between. In Table 2.5, when the number of servers doubles while fixing the number of records per server, 85% of the time increase comes from additional shuffle costs; shuffle costs affect not only the main shuffle phase, but also the partitioning phase.

To get a better understanding of shuffle costs, I ran a stand-alone shuffle benchmark in which each of a group of servers sends a fixed-size message to each other server. Figure 2.8 graphs shuffle performance in terms of throughput per server. There were four servers per machine, so the ideal throughput per server would be 25 Gbps.

There are two different potential factors that contribute to the higher shuffle time with more servers. First, as the message size decreases, efficiency drops because of per-message overheads. With 120 servers and 0.96M total records, the average message size for shuffle is about 6.7KB, which still provides a good throughput. However, with 240 servers and 1.92M total records, the average shuffle message size drops to about 3.3KB, resulting in less than 10 Gbps throughput per server. This also justifies the quadratic scaling property discussed in Section 2.5.2; when the number of servers and number of records per server are scaled simultaneously, shuffles are more likely to maintain their efficiency since the average message size for shuffle is fixed.

Second, even with large messages, throughput per server drops as the cluster size increases. It drops from 22 Gbps per server (with 40 servers) to less than 15 Gbps per server with 240 or more servers. This is because the cluster network uses flow-consistent load balancing, rather than packet-level load balancing. With large numbers of active transmissions, paths conflict in their link usage, resulting in congestion on those links and under-usage of other links. As the cluster size increases, shuffles consume a larger fraction of the core bandwidth, which makes congestion more

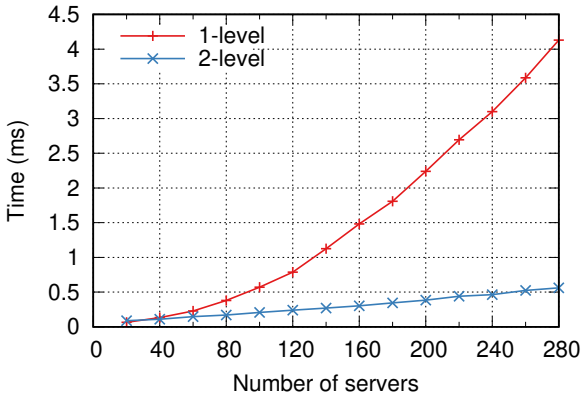


Figure 2.9: MilliSort's partitioning time as a function of cluster size with different partitioning schemes.

likely, and the high-entropy approach to shuffles cannot completely compensate. As a result, shuffles cannot harness the full bisection bandwidth offered by the network. I speculate that more granular in-network load-balancing techniques such as [74, 96] may help remove this bottleneck in the future.

2.5.6 Partitioning cost

Figure 2.9 shows the results of a stand-alone experiment that measures partitioning time as a function of the number of servers. The multi-level partitioning algorithm developed for MilliSort provides a significant peformance benefit: by the time the number of servers reaches 200, the 2-level approach is more than 5x as fast as the 1-level approach. As the number of servers increases, the partitioning time increases super-linearly for both 1-level and 2-level partitioning schemes. However, for 2-level partitioning, the increase in time is almost linear up to 280 servers (each additional server adds about 2 μs in the current implementation). A 1-level approach is too slow to harness 100 or more servers in a 1 ms budget, but a 2-level approach can easily coordinate 120 servers within 1 ms and 280 servers within 10 ms. In the best configurations for 1 ms and 10 ms time budgets, the partitioning cost only accounts for about 20% and 5% of the total time, respectively. Unfortunately, even with 2-level partitioning, the coordination cost for 280 servers is already more than half a millisecond; this prevents us from harnessing hundreds of servers in 1 ms for MilliSort. I didn't implement 3-level partitioning and beyond, but I expect that increasing the number of levels will further reduce the coordination cost for large clusters.

2.5.7 MilliSort efficiency

It might seem that flash bursts must sacrifice throughput (or efficiency) in order to operate at millisecond timescales. However, the results suggest that this need not be true. Table 2.6 shows the overall efficiency of MilliSort compared to purely local sort and ideal distributed sort. The

	Throughput (efficiency)	
	120 servers (0.84M records)	**280 servers** (26M records)
MilliSort	7172 (19.6%)	10378 (33.1%)
Ideal Distr. Sort	11272 (30.8%)	13600 (43.3%)
Local Sort	36656 (100%)	31398 (100%)

Table 2.6: MilliSort efficiency of the best configurations for 1 ms and 10 ms time budgets, compared to local sort and ideal distributed sort. Throughput numbers are in "records/ms/server". Efficiency is the relative throughput compared to local sort.

	CPU	SMT	BW/core	Throughput
MilliSort	Xeon Gold 6148 @ 2.4GHz	1	3.1	1297 (1.00x)
TencentSort	IBM POWER8 @ 2.9GHz	8	5.0	1977 (1.52x)
CloudRAMSort	Xeon X5680 @ 2.9GHz	2	2.7	707 (0.55x)

Table 2.7: Per-core throughput comparison of MilliSort, Tencent Sort [47], and CloudRAMSort [52]. "SMT" (i.e., simultaneous multithreading) is the number of hardware threads per core. "BW/core" is the average network bandwidth per core, in Gbps. "Throughput" numbers are in "records/ms/core". MilliSort's per-core throughput is computed from the 10 ms configuration in Table 2.6, while Tencent Sort and CloudRAMSort's numbers are computed from their published results [47, 52]. Tencent Sort's reported throughput is for sorting from disk to disk; it's estimated that less than half of the time is spent on disk I/O, so I double its reported throughput for a fair comparison.

ideal distributed sort represents an imaginary situation where the partitioning cost is zero and data are shuffled at full network bandwidth; this provides an upper bound on the throughput of any distributed sort. Despite operating at a 1 ms timescale, MilliSort is relatively efficient (e.g., the maximum possible throughput for distributed sorts is only about 50% higher); MilliSort is even more efficient at a 10 ms timescale.

MilliSort's throughput is also on par with state-of-the-art sorting systems [52, 47] that have much longer running times. Table 2.7 summarizes the hardware differences and presents the throughput numbers in "records/ms/core" for comparison. In particular, Tencent Sort [47] is the current record holder for the GraySort benchmark [91], and its total running time is about 100 seconds. MilliSort's throughput per core at 10 ms is only 33% lower than Tencent Sort, even though Tencent Sort's POWER8 cores have 8x the hardware threads and 1.6x the average network bandwidth of MilliSort cores.

Table 2.6 also provides insight into why distributing data at very fine granularity makes sense. Network communication is the largest source of overhead for distributed computation (the throughput of local sort is 3–5x higher than MilliSort in Table 2.6). However, most of this cost is paid immediately when scaling beyond a single machine. For example, performing a distributed sort with just two machines requires half of the data to be sent over the network. There is not much additional loss in efficiency when scaling out further. This suggests that if you can afford to distribute at all,

you can afford to distribute a lot.

2.6 Observations

This section summarizes the key observations that emerged from the MilliSort and MilliQuery experiments:

Granular data distribution. Distributing data in fine granularity allows one to harness more CPU cores and aggregated network bandwidth for faster computation and communication. The experiment also confirms that, at least for some dwarfs, it is possible to scale out quite efficiently even at millisecond timescale. As a result, I predict that future distributed data-parallel systems will need to be optimized for large scale-out architectures with smaller data per server. This will likely present interesting challenges to the design of future systems since they will be required to scale down gracefully (i.e., operate efficiently even on smaller data).

Efficient group communication is essential. Even simple dwarfs have complex communication patterns internally; all of the benchmarks except MilliQuery Q1 depend heavily on group communication. Even when carefully optimized, they account for 50%–60%, 35%–40%, and 50%–65% of the overall running time in the best configurations of MilliSort, MilliQuery Q2, and MilliQuery Q3, respectively.

Per-message overhead is critical. Network bandwidth and latency are well known to be important for the performance of large-scale systems, and they remain important for flash bursts. Flash bursts differ from traditional large-scale systems in that per-message costs are at least as important as the traditional metrics. This is because group communication primitives tend to generate many small messages when operating at small time scales with large cluster sizes.

Coordination must be structured hierarchically. It is well known in the HPC community that communication must be structured hierarchically to handle large cluster sizes. Thus, group communication primitives are commonly implemented in a tree or hypercube topology [93], so that each machine only communicates with a small number of its peers. This helps to mitigate per-message overheads and harness more aggregated network bandwidth. This principle also applies to coordination. As an example, MilliSort's partition phase is structured as a hierarchical series of distributed sorts, so that the computation doesn't become a central bottleneck.

Low-latency shuffle is challenging. Of all the group communication primitives, shuffle is the most challenging in flash bursts. There are several reasons for this. First, per-message overheads become critical since shuffles require each server to send many small messages. Unlike other primitives, a hierarchical approach to shuffles generally isn't practical because it doubles the network bandwidth utilization. Second, shuffles need to use the full network bandwidth of each server for best performance, but many networks do not provide full bisection bandwidth, especially at large scale. Even

where full bisection bandwidth is available, transient congestion can occur due to imperfect bipartite matching, either in the network itself (because of routing) or in the application. Finally, hash partitioning often results in non-trivial data imbalance (e.g., in MilliQuery Q2, the most unlucky server typically has to process 1.5x much data as the average). My experience with shuffles in the MilliSort and MilliQuery experiments motivated the rest of the work in this dissertation.

Quadratic scaling. As the time budget increases, the number of machines that can be harnessed effectively increases at least linearly. The amount of data each server can process also grows at least linearly with the time budget, so the overall dataset size grows at least quadratically with time budget. For some workloads, such as MilliQuery Q1, where the only coordination overhead is a final aggregation, the number of machines can grow exponentially with the time budget, so the dataset size grows faster than quadratically. The flip side of this is that reducing the time budget results in superlinear reductions in the number of servers and overall dataset size, which creates a lower limit on the timescale at which the system can operate efficiently. In the experiments, 1 ms appears to be tractable for all of the workloads except MilliQuery Q3. Future systems that are built on top of better networking infrastructure will likely enable these workloads to run efficiently at an even smaller timescale.

Ultimate limits of strong scaling. If one increases the number of machines while fixing the dataset size, efficiency inevitably decreases; this eventually prevents us from harnessing more machines to speed up the job. While it is well known that the theoretical speedup of a parallel program is ultimately limited by its serial portion (Amdahl's Law), this is not the limiting factor in the experiments. Instead, two other factors result in the drop in efficiency. First and foremost, coordination cost rises. This is mostly due to per-message overheads (e.g., manifested as higher shuffle costs). Coordination algorithms also take longer to run (e.g., MilliSort's partitioning time increases linearly with the number of servers, even with the recursive scheme). Second, straggler effects are more significant when there are more servers and less data per server (e.g., even in MilliQuery Q1, the overall efficiency drops about 50% when scaling from 40 servers to 280 servers).

2.7 Applicability of results

The experiments conducted earlier are subject to limitations and assumptions that may not apply to today's computing systems. This section discusses some of these factors and argue that the results will be relevant for future systems.

Is 1–10 ms the right target? 1–10 ms is not the timescale at which most people think of large-scale distributed computation today. For example, response times for users of a few hundred milliseconds are considered adequate, and it can take 100 ms just to communicate between browser and datacenter. However, applications such as AR/VR require response times of 10 ms or better,

and new edge computing offerings such as AWS Local Zones and WaveLength [6] can already provide single-digit millisecond latency between the end-user and an edge cloud. Furthermore, there are increasing numbers of applications that make real-time decisions without humans in the loop. Examples include controllers for autonomous vehicles [62] and IoT devices [40], which make decisions on the order of 10 ms, and financial applications [38, 80], for which there appears to be no lower bound on desirable latency. Flash bursts can enable these applications to run data-intensive algorithms at millisecond timescale.

Idealized experiment setup. The experiments are conducted on a bare-metal HPC cluster with no interference from competing workloads. A dedicated cluster is not economically feasible in practice; however, recent work on colocating latency-critical and batch jobs [75, 28] could be applied to achieve high CPU efficiency without hurting the performance of flash bursts. In addition, input data are assumed to be resident in memory when the experiments start. This may not be a realistic assumption today, where data is stored on flash or disk. However, future datacenter systems are likely to store data in nonvolatile memories (NVMs) with access times not far above today's DRAM [45]. Ideally, future applications will run directly on NVM-based storage servers; in the worst case an initial shuffle step will be needed to extract data from the storage servers to computational nodes.

Missing pieces. Given the tight time budget, a flash burst requires every component involved in its lifetime to support low latency. There are several elements that this work did not address, such as application loading, but there are many other projects attacking these pieces, such as storage systems [77, 24, 61, 54, 42], cluster schedulers [49], networking infrastructure [69, 50, 65, 55, 25], fast threading and dispatching [8, 81, 48, 75, 83], and lightweight virtualization [1]. Many interesting problems have yet to be solved [58] in order to create a unified flash burst infrastructure.

2.8 Summary

MilliSort and MilliQuery demonstrate that several core patterns in data analytics can run efficiently at millisecond timescales. With a budget of 1 ms, most of the patterns can harness at least 100 servers; with a budget of 10 ms, the results suggest that most of the patterns can harness at least 1000 servers. Furthermore, the results indicate that there is only a small efficiency penalty for running at millisecond timescales. Two related problems that currently limit scalability are identified: per-message costs and shuffle overheads. If future implementations can improve in these areas, it should be possible to execute large-scale computations even more efficiently, and at timescales even less than one millisecond.

However, today's systems and infrastructure were designed to work efficiently at much larger timescales and are inadequate to meet the demands of flash bursts. As a result, making flash bursts practical will require advances in many different areas across the entire computing stack. The rest of the thesis will focus on only one aspect of the problem: how to perform shuffles as fast as possible

in flash bursts. This thesis doesn't address issues such as the time required to load applications and data, or how to achieve high resource utilization in the face of short-lived computations; they are left for future work.

Chapter 3

Protocol Design

The ultimate goal of our shuffle protocol is to achieve near-optimal performance for all kinds of workloads and under various demanding conditions: an ideal shuffle protocol should scale smoothly to handle shuffles that use a large number of nodes (e.g., 1000 or more nodes) and work well for very short time intervals (as little as a few hundred microseconds), and it should be robust to interference from the environment and the underlying network topology (e.g., the underlying network may not provide full bisection bandwidth). The performance metric to focus on is the overall shuffle time, i.e., the completion time of the last message, since stragglers often have significant impacts on the overall performance of distributed applications.

Today, many people consider shuffle a solved problem, as it appears to have little opportunity for performance improvement. As a result, even though shuffle is one of the most widely used communication patterns in Big Data and HPC workloads (often referred to as all-to-all communication in the HPC community), there has been relatively little research on the scheduling algorithm of shuffle, compared to other communication patterns such as broadcast, gather, and all-reduce [94, 98, 106]. However, as our results will show, existing designs may only be optimized for certain workloads, they may not be applicable to the short time scales of flash bursts, and they can suffer from significant performance degradation in non-ideal environments.

This chapter is organized as follows. Section 3.1 will define the shuffle problem formally, present the observations that motivate the need for a new shuffle protocol, and derive the key design features of the shuffle protocol required both in an ideal environment and under more challenging conditions. Then Sections 3.2 to 3.5 will describe the protocol in detail, including aspects that are less essential for performance but nonetheless required by a complete protocol.

3.1 Motivation and key ideas

3.1.1 The problem of shuffle

Shuffle is a communication pattern that involves a cluster of nodes, where each node sends an individual message to every other node: if the number of nodes that participate in the shuffle is N, then each node needs to send and receive a total of $(N-1)$ messages. A shuffle is complete when all the messages are received in their entirety, and our goal is to minimize the completion time of the shuffle.

In general, each node is both a sender and a receiver, so the number of senders is equal to the number of receivers. However, this is not a hard requirement: some nodes may not have any data to send or receive in a shuffle. For instance, in MapReduce [21], the senders and receivers of the shuffle phase are usually two disjoint sets of physical nodes, so each node is either a sender or receiver, but not both. Note that the MapReduce example is merely a special case of the more general shuffle problem (where some of the message sizes are zero); thus, for simplicity, I will simply say that each node is both a sender and receiver from now on.

The maximum amount of data that must be sent or received by any node determines a lower bound for the overall completion time of shuffle, and the nodes that have the most data to send or receive will be referred to as the bottleneck nodes. If a shuffle workload only has a few bottleneck nodes, then any protocol that manages to fully utilize the link bandwidth of these nodes will achieve the optimal shuffle time, unless somehow it's terribly inefficient for some other nodes. However, if every node in the cluster sends and receives the same amount of data (i.e., the shuffle has a *uniform data distribution*), then every node is a potential bottleneck: an optimal protocol will have to fully utilize the link bandwidth of every node simultaneously in order to minimize the shuffle time, since any delay of a node can turn it into a straggler. As a result, shuffles with uniform data distribution present the most challenges to the protocol design, so, unless otherwise stated, this will be the default case in our future discussion (empirically speaking, if a protocol works well for uniform data distribution, it should also work well in other cases).

Another important aspect of the shuffle workload is the message size: in general, shuffle messages can vary significantly in length (some messages may even be empty). If all the messages in a shuffle have roughly the same size, the shuffle is said to have *uniform message distribution*[1]; this is often considered by many as the most common workload in practice, and we shall see that it's also the least challenging one for protocol design. In fact, almost all existing designs should achieve near-optimal performance for the case of uniform message distribution. Unfortunately, most of these designs are only optimized for this simple case, and, as we will see later, their performance is very sensitive to the variation of message sizes: even a slight difference in message sizes can cause non-negligible

[1] Uniform data distribution doesn't entail uniform message distribution: even if each node sends and receives the same amount of data, the individual messages sizes can vary a lot.

performance degradation.

I also make two assumptions about the underlying network that interconnects the nodes in the shuffle. First, I assume that every node has the same network speed. The sole purpose of this is to simplify discussion, and, as we will see later, my design is completely agnostic to the network speed of individual nodes. Second, I assume the nodes are interconnected by a datacenter network that consists of multiple levels of switches (e.g., [89]). Each node is directly connected to a Top-of-Rack switch (ToR), and each ToR is further connected to multiple switches at a higher level; these higher level switches constitute the core of the network. It's important to clarify the network topology, as the bisection bandwidth of a network is another crucial factor affecting the shuffle performance.

Ideally, if the underlying network provides full bisection bandwidth, then persistent congestion will be rare in the network core (assuming the underlying network utilizes randomized per-packet spraying to ensure load balancing across core links [22]), so the performance bottleneck in hardware is likely the link speed of individual nodes. However, full bisection bandwidth is typically not available in today's datacenter network beyond a single rack; unfortunately, large-scale shuffles will definitely span multiple ToRs. In theory, it's possible to achieve optimal performance with less bisection bandwidth if the nodes are smart enough to interleave their intra-rack and inter-rack traffic in some collective way [82]. As a result, the challenge here is to design a protocol that utilizes the bisection bandwidth wisely, and in a general way that doesn't require a priori knowledge of the underlying network topology.

3.1.2 Prior work

Today, many people consider shuffle a solved problem, and I hypothesize two reasons for the lack of more research on efficient shuffle algorithms. First, unlike many other collective communication patterns, shuffle appears to have little room for optimization: recall that every node needs to send a message to every other node, and that every message is distinct (this is why shuffle is also known as the all-to-all *personalized* communication in the HPC community), so every byte in the shuffle messages must be transmitted at least once over the network by its sender. This essentially invalidates any bandwidth reduction techniques used in other communication patterns (e.g., all-gather), or the attempt to harness more nodes to transmit the same piece of data faster (e.g., broadcast). Second, simple solutions may perform just well enough for the most common workloads so that shuffles don't become the bottlenecks of their applications. Indeed, unless an algorithm is terribly flawed, it should complete well within 2x of the optimal shuffle time normally (discussed below in Section 3.1.3). However, my study on MilliSort and MilliQuery has shown that shuffle is the most significant bottleneck that limits the scalability of flash bursts. Further, the experiments show that existing algorithms suffer from performance degradation when the message sizes are more skewed, when there is interference from the underlying environment, or when the bisection bandwidth of the network is scarce.

The most naive solution to shuffle is to hand off all the messages to the underlying network transport and let the transport schedule them with its own policy, and it's not uncommon in practice. For instance, NCCL [72], a widely used library for efficient multi-GPU and multi-node communication, takes this route. However, the biggest problem of this approach is that, without being shuffle-aware, the underlying network transport often has a very different objective to optimize (e.g., the average completion time of individual messages, message deadlines, or fairness across flows). For instance, low-latency transports such as Homa [69] schedule messages using the Shortest Remaining Processing Time (SRPT) policy, but my experiment shows that SRPT is almost the opposite of what we want when the goal is to minimize the completion time of the last message in shuffle. As a result, the performance of this approach is sub-optimal at best.

Another common approach in practice, especially for HPC applications, is to schedule the messages in a rotating and lockstep fashion [94]. For a cluster of N nodes, the algorithm completes in $(N-1)$ steps: in step k $(0 < k < N)$, node i transmits the message to node $(i+k) \bmod N$. To avoid conflicts between steps, a global barrier can be used to wait for all messages in-flight to complete before advancing to the next step. Unfortunately, this approach only works well when all the messages have roughly the same size, and are long enough to amortize the cost of synchronization in each step. Further, this approach handles the lack of full bisection bandwidth (either due to limited core bandwidth or routing inefficiency) poorly; recall that this is one of the early approaches I tried with MilliSort and the resulting performance was disappointing (Section 2.4.1).

Hadoop uses a randomized quasi-fair sharing strategy: each receiver opens connections to multiple random senders and then relies on TCP fair sharing among these flows [14]. A larger number of connections per receiver improves fairness, but it also increases the risk of incast. As a result, the number of connections is chosen empirically, and study [14] shows that 5 connections per receiver is enough to achieve near-optimal performance in their evaluation. Similar to the lockstep approach above, Hadoop's approach won't work well for workloads with non-uniform message distributions.

Orchestra [14] proposes an optimal scheme called Weighted Shuffle Scheduling (WSS) for homogenous networks. WSS assigns each message a weight that is equal to its (remaining) data size and relies on an underlying network transport to properly divide the bandwidth of the bottlenecks (i.e., sender uplinks, receiver downlinks, and links between switches in the network core) in a weighted fair sharing fashion. As long as the underlying network transport is decentralized and successfully implements the weighted fair sharing policy, WSS is decentralized and optimal; this can be proved by showing that at least one network link is fully utilized throughout the shuffle.

Orchestra did not provide a concrete implementation of such an ideal transport though; instead, it implemented a prototype by building on TCP's AIMD fair sharing behavior, and opening a variable number of connections for each message, proportional to its data size. Emulating flow weights using multiple connections per flow has several severe drawbacks. First, it is impossible to specify the flow weights accurately without using a large number of connections. Second, having a large

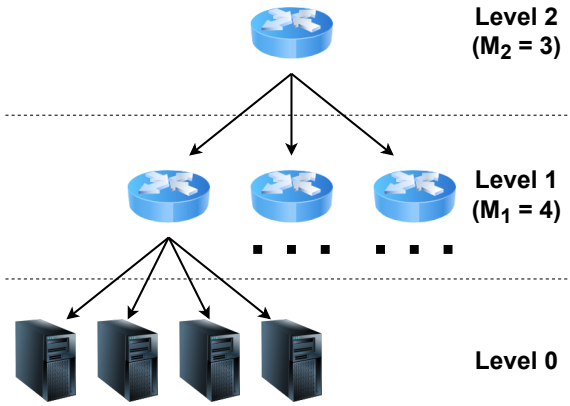


Figure 3.1: A three-level idealized fat tree topology.

number of connections creates a huge challenge to enforcing the fair sharing behavior. For example, Swift [55] shows that, with a 5000-to-1 incast, the flow rates fluctuate significantly (within 2x of the optimal) even over the course of 10s of seconds. Finally, opening connections is usually an expensive operation, which makes it unsuitable for flash bursts (one cannot simply reuse connections because we don't know how many connections will be opened to a server before the shuffle starts). There are numerous datacenter transport protocols that implement fair sharing (approximately) [36, 55, 60]. Using any one of them will likely result in better performance than the default TCP algorithms of the OS kernel (e.g., Reno, CUBIC, ...); however, all of them will suffer from the drawbacks described above.

Thus, a more promising approach would be to employ transport protocols that support weighted fairness directly. Crowcroft et al. [16] is probably the first to present a weighted fair TCP, and some more recent examples include NetShare [57], Seawall [88], and NUMFabric [70] (there are likely more, since this is an extensively studied area). It's beyond the scope of this dissertation to evaluate how accurately these protocols allocate rates in practice to match the theoretical target, especially at short timescales relative to the network RTT; instead, I will later show that there is a much simpler way to implement WSS faithfully in a datacenter network without using a general-purpose network transport that must handle flows arriving dynamically.

Varys [15] proposed an algorithm called Minimum-Allocation-for-Desired-Duration (MADD) that generalized WSS to handle non-homogeneous links. MADD proceeds in two steps. First, it computes the minimum completion time, Γ, of the shuffle by calculating the minimum time to finish data that flow through each link and taking the maximum value. Second, it attempts to set the rate of each flow explicitly so that all flows finish at exactly time Γ. However, this generalization is fundamentally more centralized than WSS, since it requires knowing the link speeds and flow sizes in order to set the rates properly at senders.

There are also some interesting works [82, 92] that study optimal shuffle schedules in the case of

limited core bandwidth. In particular, Prisacari et al. [82] derive a lower bound on the core bandwidth required to avoid increasing the optimal shuffle time: given an *idealized* fat tree topology [99] of $L+1$ levels, with 0 being the level of leaves (i.e., servers) and L that of the single root (in practice, fat tree topologies are often built from commodity switches with a fixed number of ports, so they don't have a single root), if each server has 1 unit of bandwidth to its ToR, then the minimum required bandwidth of a data link between level $l (l > 0)$ and $l+1$ is $M_1 \cdot M_2 \cdot \cdots \cdot M_l - \frac{M_1 \cdot M_2 \cdot \cdots \cdot M_l}{M_{l+1} \cdot M_{l+2} \cdot \cdots \cdot M_L}$, where M_i is the number of children of each node at level i. Figure 3.1 shows an example of a three-level idealized fat tree.

This is an exciting theoretical result because it suggests that optimal shuffle times can be achieved without full bisection bandwidth. Consider, for example, a three-level fat tree, as in Figure 3.1: the minimum required bandwidth between a ToR and the core switch is $M_1 - \frac{M_1}{M_2}$ according to this result (or, equivalently, the blocking factor is $M_1/(M_1 - \frac{M_1}{M_2}) = \frac{M_2}{M_2 - 1}$). Thus, if a shuffle consists of the servers of exactly two racks (i.e., $M_2 = 2$), then the aggregate inter-rack bandwidth only needs to be 50% of the aggregate intra-rack bandwidth in order to achieve the optimal shuffle time (assuming uniform data distribution). Intuitively, this is reasonable because each node only needs to transmit roughly half of its data out of the rack, so a smart algorithm may come up with a schedule that always has half of the nodes transmitting cross-rack data, which prevents congestion at the core.

However, while being intellectually fascinating, the algorithms proposed in prior works have several severe drawbacks. First, they are complex enough to require a central scheduler. Second, they require knowing the underlying network topology a priori, which is usually not possible in today's cloud environment. Finally, they cannot be easily applied to irregular workloads which have non-uniform data or message distribution.

3.1.3 Optimal shuffle scheduling

In a nutshell, scheduling a shuffle is all about matching the uplink bandwidth of senders with the downlink bandwidth of receivers to minimize the completion time. Thus, this problem may appear superficially similar to (repeatedly) finding the maximum matching within a bipartite graph which consists of senders and receivers as vertices. Figure 3.2 shows a shuffle among a cluster of 4 nodes.

However, such a simplistic mental model based on bipartite matching is actually not very helpful for several reasons. First, it doesn't consider the existence of buffers in the network, which allows multiple senders to send to a single receiver simultaneously (the excessive packets can be queued in the buffer to keep the receiver busy later). Second, it also doesn't consider the possibility that each sender can send to multiple receivers simultaneously. Third, scheduling based on maximum bipartite matchings implies a greedy algorithm that only optimizes for the aggregate bandwidth utilization at the current time, but this, as we shall see later, will often result in sub-optimal scheduling globally. Finally, computing bipartite matching will likely involve some kind of centralized scheduler, but this will be too expensive to be practical, since the matching between senders and receivers often needs

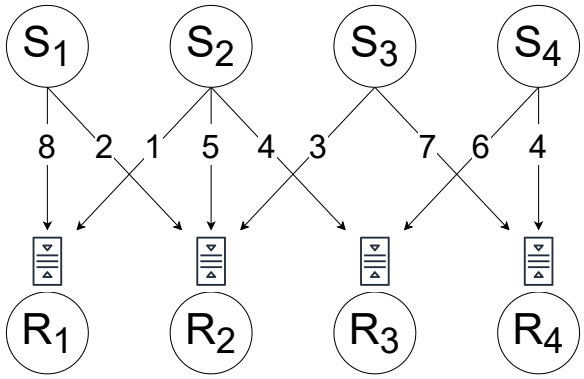


Figure 3.2: Graph illustration of a shuffle with four nodes (four senders and four receivers). The weights of the edges denote the lengths of the messages. In this example, the shuffle happens to have uniform data distribution (each node sends and receives roughly 10 units of data). A queue is attached to each receiver that represents the packet buffer at the receiver's ToR (this queue may be omitted for simplicity in future graphs).

to change rapidly over time in response to the dynamic environment.

In the rest of this section, I will derive four key ideas to achieving optimal shuffle scheduling in an ideal environment from first principles. Sections 3.1.4 and 3.1.5 then continue to introduce the design features built on these key ideas from the perspectives of senders and receivers, respectively. Finally, Sections 3.1.6 and 3.1.7 will further extend the protocol with a few more features necessary to achieving good performance in non-ideal environments.

The first key idea is that (bottleneck) senders should be kept as busy as possible. This is obvious because every bit of bandwidth wasted at the sender uplink will delay the completion time of that sender. In fact, if we assume an infinite buffer space in the switch, then any schedule that keeps the senders busy at all times is guaranteed to be within 2x slowdown of the optimal schedule. Figure 3.3 shows such an example. Intuitively, in the worst case, it could take 1x time for the senders to inject the data into the network, and then no more than another 1x time for the receivers to retrieve them, while these two phases can be completely overlapped in an optimal schedule; hence the 2x slowdown. Finally, it's worth noting that this key idea applies to non-bottleneck senders too, since they could become the new bottlenecks if they waste too much bandwidth or, more likely, are randomly delayed too much by interference from competing workloads (discussed later in Section 3.1.6). Said another way, if buffer overflow is not a concern, then each sender should use its uplink bandwidth sooner rather than later (it's easy to prove that this yields a schedule that is at least as good as before; the only downside is increased buffer usage).

The second key idea is that (bottleneck) receivers should also be kept as busy as possible. As shown in Figure 3.3, just keeping the senders busy is not enough because it doesn't prevent a bad schedule from producing a lot of *persistent* incasts; this results in network congestion at some receivers, and wasted downlink bandwidth at other receivers. As a result, the senders must also

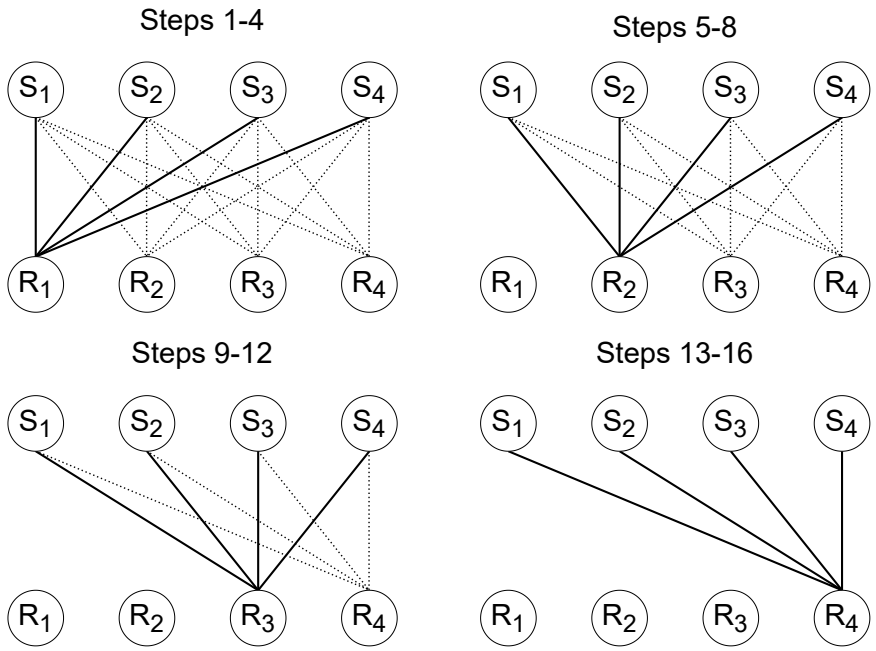


Figure 3.3: A pathological shuffle schedule. Each message has four units of data, and each node is capable of sending and receiving one unit of data per step. Solid lines denote messages that are scheduled to be transmitted; messages that are fully transmitted are removed from the graph. Ignoring the communication latency, an optimal schedule can complete in 16 steps, while the pathological schedule will need 28 steps (R_4 needs 16 steps to receive all incoming data, but it won't receive any data until after step 12), or roughly a 2x slowdown.

spread out their outgoing traffic (over time) to avoid such pathological schedules. For example, Figure 3.4 shows a much better schedule when senders divide their uplink bandwidth evenly among receivers. It's interesting to notice that transient incasts are much less damaging, since each of them will build up a small queue at the receiver, which in turn reduces the likelihood of wasting bandwidth at this receiver in the near future.

Under the assumption of infinite buffer space, sticking to the two key ideas above is sufficient to produce near-optimal shuffle schedules. However, in practice, any scheduling algorithm must employ some sort of congestion control mechanism to avoid buffer overflow, so, in general, it's not possible to keep all senders busy at all times.

As a result, the third key idea is that the congestion control should be done in a way that senders which are more demanding on bandwidth are throttled less. Figure 3.5 provides a concrete example. To avoid buffer overflow, the average rate of incoming data for each receiver should not exceed 1 unit per step. If both receivers throttle the incoming messages evenly (i.e., each sender transmits at the rate of $\frac{1}{3}$ units of data per step), then all senders except S_3 will finish after $3x$ steps. However, S_3 still has $2x$ units of data left to transmit, and it will take another $2x$ steps to finish them (now

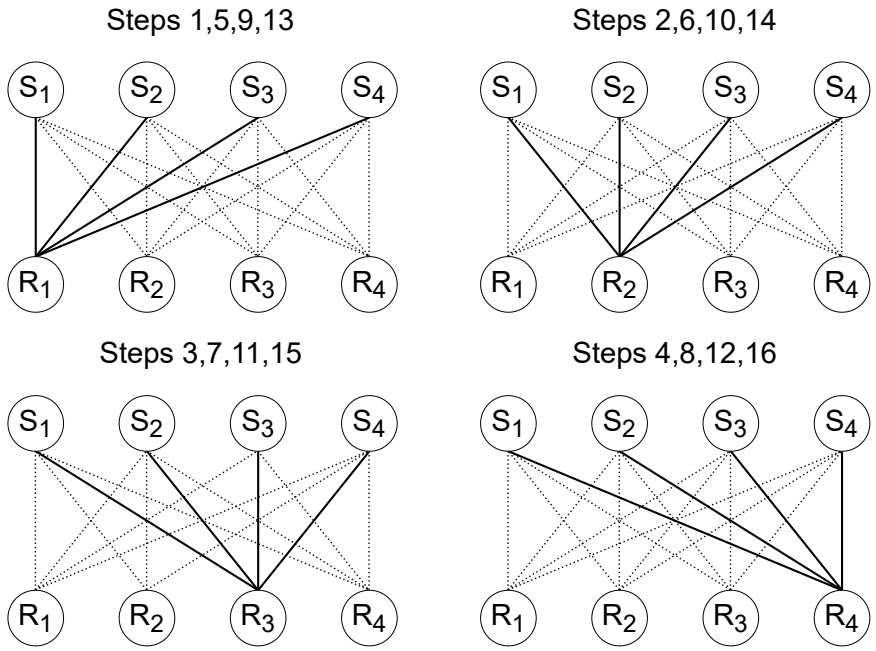


Figure 3.4: A much better shuffle schedule than the one in Figure 3.3. The workload is the same. However, each node chooses its target receiver in the same round-robin fashion. This schedule is near-optimal despite incurring a lot of transient incasts: R_4 will receive all incoming data in 19 steps (i.e., 3 more steps after all senders complete), while the optimal completion time is 16 steps.

S_3 can transmit at full speed); thus, the total completion time is $5x$ steps. Now consider a different throttling strategy: if the messages from S_3 are throttled to $\frac{1}{2}$ units of data per step, while other messages to $\frac{1}{4}$ units of data per step, then all messages will complete at the same time after $4x$ steps. The major observation here is that the first strategy causes every sender but S_3 to finish too early, but S_3 alone is not enough to saturate R_1 and R_2 simultaneously. Thus, S_3 should receive a larger share of bandwidth because it has more outgoing data than any other sender; otherwise, it would become a straggler.

It's also worth noting that if this shuffle were scheduled based on computing maximum bipartite matching, then it could end up with an even worse schedule where S_1 and S_5 finish after x steps, S_2 and S_4 finish after $2x$ steps, and S_5 finish after $7x$ steps. In other words, maximizing the aggregate bandwidth at every step doesn't necessarily minimize the total completion time in the end.

Finally, flash bursts require that scheduling must be done in a decentralized way: senders and receivers each need to make their own decisions independently, without relying on a global authority. At the same time, these local decisions must produce a result that is nearly optimal from a global standpoint. This is because flash bursts run at large scales but for very short time intervals, so any central bottleneck can limit the scalability of flash bursts: e.g., maintaining a global state of shuffle requires frequent network communication between the central scheduler and the rest of the cluster,

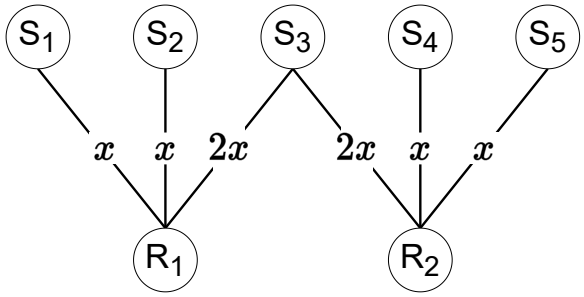


Figure 3.5: A shuffle example to illustrate how the choice of throttling affects the completion time. S_3 is the only bottleneck sender because it has $4x$ units of data to transmit; both R_1 and R_2 are the bottleneck receivers because each of them also has $4x$ units of data to receive. The value of x is assumed to be very large.

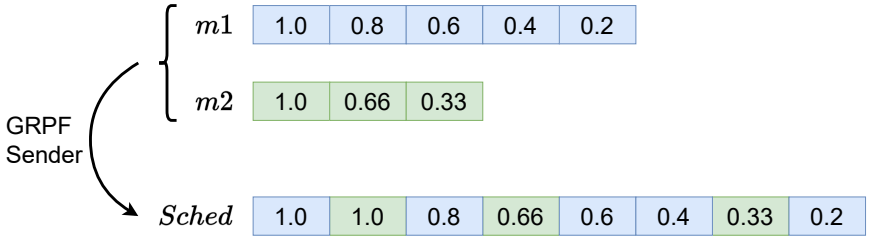


Figure 3.6: Sender-side scheduling with GRPF. There are two outgoing messages m_1 and m_2 that have 5 and 3 data packets, respectively. Each box represents one data packet, and the value inside each box denotes the remaining processing fraction of the message before sending that packet. The resulting schedule at the bottom of the graph shows the order in which the packets will be sent out.

which puts a significant burden on the central scheduler to process all the network messages (not to mention the costs to compute the schedule). Thus, a central scheduler is fundamentally ill-suited to flash bursts.

3.1.4 Sender-side scheduling

The topmost design goal of the sender-side scheduling mechanism is for each sender to spread out its uplink bandwidth independently while still keeping all receivers busy (recall the second and fourth key ideas in Section 3.1.3). To achieve this goal, I propose a local policy called *Greatest Remaining Processing Fraction* (GRPF). The remaining processing fraction of an outgoing message is the ratio between the bytes yet to be sent and the total bytes of that message, and the GRPF policy uses this information to prioritize the sender's bandwidth: when it is time for the sender to transmit the next data packet, it will transmit from the message that has the largest remaining fraction (ties are broken arbitrarily). Figure 3.6 shows an example *packet schedule* (the values in the schedule are the *scheduling priorities* assigned to the packets) produced from the GRPF policy.

The sender needs to saturate its uplink bandwidth without buffering a lot of packets in its egress

queue. Ideally, the sender would be able to implement this by monitoring the length of its egress queue: e.g., if it takes the end host 1 μs to transmit one full data packet, then the sender can check the size of the egress queue every 1 μs and send out more packets if the queue size drops below some threshold. I will later refer to this simple scheme as the *sender-driven GRPF algorithm*.

An important observation of Figure 3.6 is that GRPF actually implements the so-called *pro rata rate allocation* scheme on the sender side: i.e., each outgoing message will receive a share of uplink bandwidth that is proportional to its (remaining) length. This scheme has a very nice property that if every node has the same amount of data to send and receive, then the sending rates of all incoming messages of any receiver will add up exactly to its downlink speed (there will be no congestion or idle downlinks), and every message will finish at precisely the same time. As a result, for shuffles with uniform data distribution, a purely sender-driven scheduling algorithm based on GRPF is already sufficient to achieve optimal performance.

But what if the shuffle doesn't have uniform data distribution and some receiver has more data to receive (i.e., the bottleneck receiver) than others? Fortunately, even in this case, the pro rata rate allocation scheme still guarantees that the downlink of the bottleneck receiver will be fully utilized; more precisely, its downlink will be overloaded at first, but then its incoming messages will gradually be throttled to match the link speed (this will be discussed in Section 3.1.5). To see why this happens, consider the aggregate sending rates of all the messages destined to this bottleneck receiver: if each incoming message m uses exactly $\frac{m.length}{total\_in\_bytes}$ of the link capacity, where $total\_in\_bytes$ is the total incoming bytes of the receiver, then the sending rates of all messages will sum up to exactly the link speed because $\sum_{\forall m} m.length = total\_in\_bytes$. However, the initial sending rate of any message m as computed by its sender will be equal to $\frac{m.length}{total\_out\_bytes(m)}$ instead, where $total\_out\_bytes(m)$ is the total outgoing bytes of m's sender. Recall that the receiver is the bottleneck, so $total\_out\_bytes(m)$ must be smaller than $total\_in\_bytes$, and thus the resulting aggregate sending rates must exceed the link capacity of the receiver.

Unlike the classic Water Filling algorithm and its derivatives [9, 18, 17], the GRPF policy can be implemented on each node efficiently without a central scheduler. As a bonus, the GRPF policy also naturally adapts to the ever-changing remaining fractions of messages, so straggler mitigation already comes for free: in practice, an outgoing message can fall behind its peers for many reasons (this will be discussed later), and GRPF ensures that this message will always get priority once it's ready to accept more bandwidth.

3.1.5 Receiver-driven rate control

The GRPF policy on the sender side discussed earlier provides a good starting point to achieving optimal shuffle scheduling. However, a purely sender-driven approach is not sufficient: without some sort of feedback loop involving the receivers, the senders cannot detect congestion and prevent buffer overflow by themselves. For example, in Figure 3.5, the switch buffers in front of the downlinks of R_1

and R_2 will quickly overflow if the senders simply transmit at full speed independently. Though this example is certainly a pathological case, the same problem exists in almost every shuffle workload. As discussed in Section 3.1.4, if a shuffle has perfect uniform data distribution, then GRPF can assure a perfect matching between the senders' and receivers' bandwidth, and there will be no congestion at all; however, even if there is just one sender that has slightly less total data than others, the sending rates of the messages from this sender will be a little bit too high to maintain this nice equilibrium (buffer overflows will occur given enough time).

Packets are dropped when the switch buffer is full. However, unlike many traditional workloads, flash bursts can be extremely sensitive to packet drops, since flash bursts may only run for a very short period such as a few milliseconds, which is on the same order of magnitude as the timeout for lost packet detection; thus, even a single dropped packet can increase the completion time of flash bursts significantly.

As a result, it's crucial to design the shuffle protocol that eliminates buffer overflow (completely) via some rate control mechanism. In addition, as discussed in Section 3.1.3, the rate control should allocate larger shares of downlink bandwidth to longer messages in order to avoid stragglers. In the rest of the section, I will introduce the receiver-driven rate control scheme, and then discuss how it can be used to achieve the two design goals above.

Existing mechanisms for rate control in datacenters usually fall into one of three categories: (i) centralized scheduling [78], (ii) sender-driven scheduling [68, 63], and (iii) receiver-driven scheduling [69, 36]. Centralized scheduling is fundamentally ill-suited to flash bursts as it creates a central bottleneck. Sender-driven approaches typically use indirect signals, such as RTT, or explicit congestion notification (ECN) to infer the buffer occupancy at the receivers, and then use that information to adjust the sending rate accordingly. Though it's possible to eliminate buffer overflow completely with a carefully-engineered sender-driven scheme, the receiver is better positioned for this task, since it knows exactly the set of incoming messages competing for bandwidth on its downlink. As a result, I will focus on receiver-driven rate control from now on.

In a receiver-driven rate control scheme, the receiver typically uses explicit *grants* to schedule incoming data packets from the senders: a grant indicates the total amount of data in a message that can be transmitted by its sender, and the senders transmit packets in response to grants. Usually, a sender cannot transmit data without permission from its receiver. However, there is one exception: in order to reduce the latency of small messages, each sender is allowed to transmit the initial bytes of each message blindly; these bytes are referred to as *unscheduled data*. For simplicity, the following discussion assumes that each grant invites the sender to transmit exactly one *full* data packet (i.e., the maximum packet size that can be transmitted without fragmentation in the network), so there is a one-to-one relationship between grants and (scheduled) data packets. Finally, as grants are tiny packets, I assume they will be assigned a higher network priority than data packets by default, which allows them to bypass the output queues of the switches (provided that the corresponding output

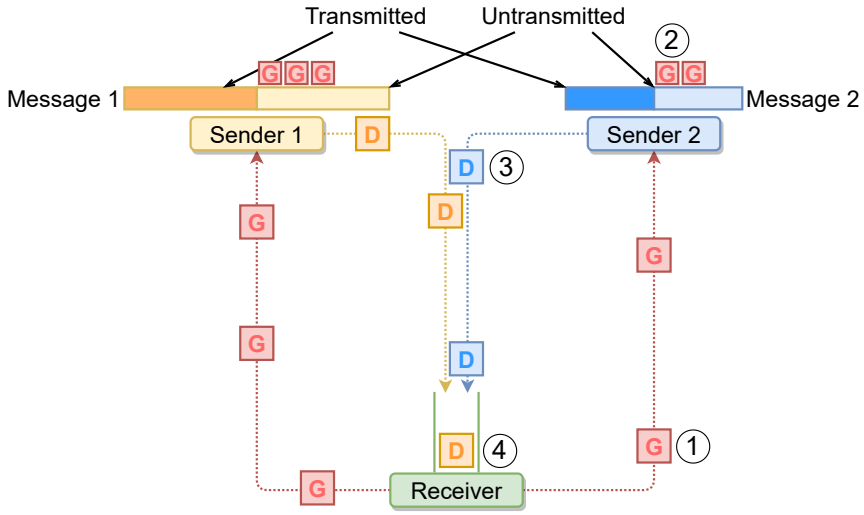


Figure 3.7: Lifetime of an outstanding grant. An outstanding grant is always in one of the following four stages: ① outgoing grant on the wire, ② granted data packet (waiting to be sent) at the sender, ③ incoming data packet on the wire, and ④ incoming data packet queued at the ToR switch. Note that this graph omits grants that may be queued at the sender's ToR switch, since grants can bypass the queues by default.

ports are properly configured to honor the network priorities).

A grant is said to be *outstanding* once it has been sent out by the receiver; Figure 3.7 illustrates the lifetime of an outstanding grant. It's easy to prevent buffer overflows in a receiver-driven scheme: by enforcing a maximum number of outstanding grants any receiver can have at any time, receivers can limit the incoming packets queued at the ToR switch, since it always takes one grant to receive one data packet (modulo unscheduled packets [1]). The maximum queue length is only achieved when all outstanding grants are turned into data packets queued at the switch.

Further, to avoid wasting bandwidth, the number of outstanding grants must be large enough to cover the round-trip times between nodes (including the software overheads of packet processing on both ends). Consider the example of one sender and one receiver: if the sender wants to transmit at full speed, there needs to be at least one RTT worth of outstanding grants between the two nodes; this minimum amount of packets is referred to as *RTTpackets*. This can be extended to the case of multiple senders: in order to keep its downlink busy, the receiver needs to have at least RTTpackets outstanding grants across all senders. And if the maximum outstanding grants of a receiver is larger than RTTpackets, the receiver is said to *overcommit* its downlink. Similarly, a sender also needs at least RTTpackets of incoming grants across all receivers in order to keep its uplink busy.

Below I present the basic shuffle protocol that combines the sender-driven GRPF algorithm and

[1] Although too many unscheduled packets could lead to buffer overflow at startup, it can be easily fixed by properly limiting the amount of unscheduled data (e.g., capping the total unscheduled data per sender).

receiver-driven rate control. In order to prevent straggler senders, the receiver also implements the pro rata rate allocation scheme [1]. This is also achieved by the GRPF policy: when there are multiple incoming messages that have not been fully granted, the receiver will grant to the message that has the largest remaining fraction yet to be received. At startup, each sender will send out one special packet to every receiver that contains only the size of the message to that receiver (no unscheduled data). Upon hearing from all its senders, the receiver will send out RTTpackets outstanding grants immediately and attempt to keep the amount of outstanding grants unchanged during the shuffle; thus, except at startup, the receiver only sends one grant for each incoming data packet. The sender-side algorithm remains largely the same. The only change is that now the sender cannot send a data packet before it has been granted. As a result, when it is time to send the next packet, the sender have to search the packet schedule to find the first granted packet.

With this basic protocol, the total number of grants that a receiver sends to any sender is always proportional to the size of that sender's message. Said another way, the granted fractions of all incoming messages are always roughly the same throughout the shuffle, regardless of the (remaining) message sizes.

Experiments show that this basic protocol actually performs quite well in an ideal environment regardless of the shuffle patterns. In summary, this simple protocol uses the GRPF policy to schedule both the data packets and grants; it limits the maximum outstanding grants at each receiver to avoid buffer overflow; and it maintains RTTpackets outstanding grants at each receiver to keep the senders busy. Finally, it is easily implementable without a central scheduler. In the next two sections, we will see how various non-ideal environments introduce new challenges to the protocol design.

3.1.6 Interference

An important motivation of my work is to design a shuffle protocol that performs well in today's cloud computing environment. In contrast to typical HPC workloads that run on dedicated nodes of a supercomputer, applications in the cloud rarely run in a clean and isolated environment. In order to improve hardware utilization and reduce operating cost, datacenter operators commonly colocate different applications (e.g., latency-sensitive and batch jobs) on the same server. For instance, 40% of the machines in a Google compute cluster run 10 or more applications [105]. Since flash bursts are extremely bursty (they can stay idle most of the time, and then scale to hundreds of nodes for only a few milliseconds), they will need to be colocated with other applications in order to be economically practical. This presents significant challenges to the design, since it requires the protocol to (i) react quickly to the interference to better utilize the bandwidth, and (ii) mitigate the stragglers that may arise due to the interference. For this work, I consider two major types of interference: application preemption and link bandwidth variation.

[1] More precisely, the receivers are allocating their downlink bandwidth; for simplicity, I am using the same terminology on both the sender and receiver sides.

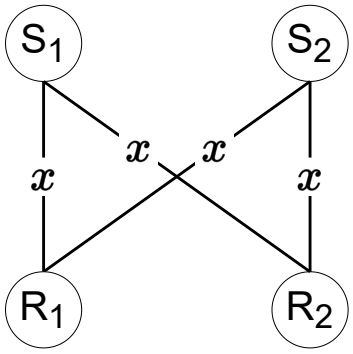


Figure 3.8: Stalled receivers reduce the amount of incoming grants at the senders. There are two senders and two receivers, and all the messages have the same size. If one of the receivers stops, the incoming grants of both senders will drop from RTTpackets to 0.5 RTTpackets.

Application preemption is the direct result of multiplexing CPU cores among applications. When multiple applications are competing for the CPU cores, the shuffle application may be temporarily descheduled at the discretion of the OS scheduler; in the meantime, the application cannot transmit or receive any packet until it's scheduled again. It's worth noting that the NIC will continue to receive packets and place them in the kernel (or in some application-provided packet buffers if kernel-bypass networking is used); these packets can be processed by the application immediately after it recovers.

The second major type of interference is bandwidth variation of the network links, which occurs when multiple applications compete for network bandwidth. Though the application is guaranteed a certain amount of network bandwidth on average, the actual bandwidth available may be fluctuating over time. As a result, random senders may appear to be slow temporarily over the course of a shuffle because some links on their data paths happen to be congested.

Unfortunately, the basic protocol described so far doesn't operate efficiently under interference: if some nodes stop working (or just appear to be slow) for an extended period of time, it could cause other nodes to under-utilize their links (when a node stops, its bandwidth will be wasted inevitably, but an ideal protocol would strive to keep other nodes busy nonetheless). Below I will elaborate two problems caused by stalled senders and receivers, respectively.

First, when a sender stops responding to grants, its receivers will nonetheless continue to pile up grants against this sender until all outstanding grants are stuck, as there is currently no limit on the number of outstanding grants issued to a sender. At this point, the receivers are blocked, and their downlinks have to sit idle until the stalled sender recovers. For example, consider a receiver with two incoming messages of equal size, sent from S_1 and S_2 respectively. Because of the GRPF policy, the receiver will always alternate its outgoing grants between S_1 and S_2. Thus, even if S_1 stops, one grant will be sent to S_1 for every two data packets arriving from S_2 until all outstanding grants belong to S_1.

Second, if a receiver stops sending out grants, its senders will have to under-utilize their uplinks,

since the number of incoming grants won't be sufficient to cover the round-trip times. Consider the shuffle example in Figure 3.8 with two senders and two receivers. Suppose both S_2 and R_2 stop, as discussed earlier, all outstanding grants of R_1 will eventually get stuck at S_2 in the current protocol; however, even if R_1 somehow knows not to send any more grants to S_2 right after it stops, half of the outstanding grants of R_1 are already committed to S_2, so there can be at most 0.5 RTTpackets grants in circulation between S_1 and R_1. Thus, both S_1 and R_1 have to run at 50% link utilization, even though S_1 could have transmitted to R_1 at full speed given enough grants.

Two important observations can be made based on the examples above: (i) receivers must not grant to unresponsive senders indefinitely, and (ii) maintaining sufficient incoming grants is the key to keeping the senders busy even if some receivers stop granting. A straw man solution would be to overcommit the receiver downlinks (e.g., increase the maximum outstanding grants to 2 RTTpackets) and introduce a constant limit on the outstanding grants issued to each sender (incoming message). With this modification, after a sender stops responding to grants, its receivers will stop granting to this stalled sender because they are constrained by the per-sender outstanding grant limit; now if some other sender decides to allocate more uplink bandwidth to one of these receivers, this receiver will be able to accommodate the sender gladly by issuing more grants in return.

However, there is a severe (but non-obvious) flaw in this design: the receivers now can no longer implement the pro rata rate allocation among their senders effectively (the receiver-side GRPF policy was sufficient before the introduction of the constant limit on per-sender outstanding grants). Consider again the shuffle in Figure 3.5. If the maximum outstanding grants is set to 3 RTTpackets and the constant limit on per-sender outstanding grants is RTTpackets, the two messages of S_3 will start being transmitted at 0.5 units of data per step, while the other ones at 1.0 unit of data per step (they are the only outgoing messages of their senders). The aggregate sending rates exceed the downlink capacity of the receivers, so the senders will eventually be throttled (packets will be queued at the receiver's downlink, which will reduce the number of grants in circulation to RTTpackets); when that happens, the sending rates of all messages will converge to the same 0.33 units of data per step (a fair sharing scheme is sub-optimal as discussed earlier). It turns out that this is a side-effect of the new constant limit on per-sender outstanding grants, which is essentially a fixed-size flow control (sliding) window for each incoming message. Recall that, in window-based rate control schemes like TCP, there is a well-known relation between the sending rate of a message and its sliding window size: if the window size is S, then the maximum sending rate of the message is $\frac{S}{RTT}$. In this example, since all sliding windows are of the same size, the message sending rates upon convergence must also be equal (when the system stabilizes, all incoming packets of a receiver will experience the same queuing delay at the ToR switch; hence the same RTT in the formula above).

Solving this problem requires using the per-message sliding windows in a slightly different way than usual: rather than using the absolute size of a window to limit the sending rate of a message, I use the relative sizes of the windows to enforce the relative sending rates among messages. Thus, to

implement the pro rata rate allocation scheme, one can simply set the per-message sliding window in proportion to the remaining sizes of the incoming message; this technique will be referred to as *pro rata sliding windows*. Below I present a correctness proof for a simple case involving two senders; curious readers may extend the proof to more senders.

Theorem 3.1.1. *Given two incoming messages m_1 and m_2 that are going to be throttled by the receiver, pro rata sliding windows ensure that the shares of downlink bandwidth they receive will eventually converge to the pro rata rate allocation scheme.*

Proof. For convenience, I assume the receiver downlink has 1 unit of bandwidth. Further, define the following symbols:

- r_i: the initial sending rate of message m_i, in units of bandwidth.
- r'_i: the final sending rate of message m_i upon convergence, in units of bandwidth.
- w_i: the sliding window size of message m_i.
- $RTTpackets(m_i)$: the round-trip time (excluding queuing delay) between the receiver and the sender of m_i, measured in the number of full data packets (i.e., the bandwidth-delay product)

The precondition states that (i) the aggregate sending rate exceeds the downlink capacity initially, and (ii) it will be throttled eventually to match the downlink speed:

$$r_1 + r_2 > 1 \tag{3.1}$$

$$r'_1 + r'_2 = 1 \tag{3.2}$$

Further, the receiver cannot speed up the incoming messages (it can only throttle them):

$$\begin{aligned} r_1 > r'_1 \\ r_2 > r'_2 \end{aligned} \tag{3.3}$$

When m_i is throttled, its sending rate is limited by how fast the grants arrive at the sender, so all granted packets of m_i must have been sent out (in Figure 3.7, there will be no grant in stage ②). In addition, since the number of outstanding grants of m_i is equal to its sliding window size w_i, the following equation of w_i holds upon convergence:

$$w_i = pkts\_in\_flight(m_i) + queued\_data\_pkts(m_i) \tag{3.4}$$

where

- $pkts\_in\_flight(m_i)$ is the number of grants and data packets on the wire, and
- $queued\_data\_pkts(m_i)$ is number of data packets queued in the receiver's ToR switch.

If m_i receives the full downlink capacity, then $pkts\_in\_flight(m_i)$ will be equal to $RTTpackets(m_i)$; however, the sender of m_i is only transmitting at the rate of r'_i, so:

$$pkts\_in\_flight(m_i) = r'_i \times RTTpackets(m_i)$$

Upon convergence, the arrival rate of incoming data packets is equal to the downlink speed, so the queue length, L, at the receiver's ToR must be a constant. Since the queue is FIFO, the number of data packets of m_i is given as follows:

$$queued\_data\_pkts(m_i) = r'_i \times L$$

Thus, Equation 3.4 can be simplified to:

$$\begin{aligned} w_1 &= r'_1 \times (RTTpackets(m_1) + L) \\ w_2 &= r'_2 \times (RTTpackets(m_2) + L) \end{aligned} \tag{3.5}$$

Combining equations 3.2 and 3.5, we have:

$$\frac{r'_1}{r'_2} = \frac{w_1 \times (RTTpackets(m_2) + L)}{w_2 \times (RTTpackets(m_1) + L)} \tag{3.6}$$

If the round-trip times in the cluster are the same (i.e., $RTTpackets(m_1) = RTTpackets(m_2)$), then equation 3.6 can be further simplified to:

$$\frac{r'_1}{r'_2} = \frac{w_1}{w_2}$$

That is, the ratio between message sending rates is equal to the ratio between sliding window sizes. □

One caveat in the proof is the assumption of a uniform RTT within the cluster. However, this is not true for large clusters: the RTT between nodes within the same rack is much smaller than the RTT of arbitrary two nodes within the datacenter. Equation 3.6 shows that senders that are farther away from the receiver (i.e., larger RTT) will receive less bandwidth than deserved. In theory, one could take RTTs into account in the formula when computing sliding window sizes, but this requires knowing RTTs a priori. Fortunately, the pro rata sliding window scheme is very adaptive in nature (discussed below), so the problem of non-uniform RTTs is usually not a concern (Section 3.1.7 will discuss more problems related to RTTs in the context of limited core bandwidth, but they are caused by queuing delays instead).

As a side note, overcommitment and pro rata sliding windows together essentially achieves the so-called *weighted max-min fair sharing* scheme: weighted fair sharing is merely a synonym for pro

rata rate allocation (i.e., the fair share of downlink bandwidth of an incoming message is proportional to its remaining size), and "max-min" indicates that each message should at least get its fair share of downlink bandwidth unless its sender cannot or decides not to use up the share (e.g., the sender may be preempted or have different priorities), in which case the unused downlink bandwidth can (and should) be assigned to other senders. In a nutshell, the key goals of weighted max-min fair sharing are *fairness* and *efficiency*, and they are achieved by pro rata sliding windows and overcommitment, respectively. For example, after a sender is preempted, its receiver will grant more to the active senders that are willing to transmit faster (i.e., efficiency); further, when the stalled sender recovers, it will regain its fair share of downlink bandwidth, and the other senders will be throttled accordingly (i.e., fairness).

One last issue of interference is that it can slow down messages randomly and create stragglers. Fortunately, both the GRPF policy and pro rata sliding windows are already well-suited to straggler mitigation. Since the sliding window size is always proportional to the remaining size of the message, when a message slows down due to interference, it will naturally receive a larger quota of downlink bandwidth in case it is ready to catch up later.

In summary, overcommiting receiver downlinks puts sufficient grants in circulation in case some receivers stop sending grants; this allows senders to be kept busy by redirecting their uplink bandwidth away from stalled receivers. Correspondingly, per-message sliding windows prevent receivers from committing all outstanding grants against stalled senders, effectively allowing them to reassign their downlink bandwidth to active senders. Finally, pro rata sliding windows are the key to upholding the optimal pro rata rate allocation scheme, when each message has its own flow control window.

3.1.7 Limited core bandwidth

Unfortunately, existing datacenter networks usually don't provide full bisection bandwidth at large scales. For example, Google uses 2:1 oversubscription in several generations of its datacenter network [89] (i.e., the aggregate uplink bandwidth of the ToR is only 50% of its aggregate downlink bandwidth).

The most direct impact of oversubscription is that the shuffle throughput is now limited by the core bandwidth of the network, so it's no longer possible to shuffle at the line rate of end hosts. Further, as the bottlenecks shift from end hosts to the core of the network, congestion can now occur not just at the downlinks of the ToR, but also at the links between the ToR and the core switches. Figure 3.9 illustrates the congestion hotspots that can occur in a realistic datacenter network.

When congestion occurs primarily at the edge of the network, it's sufficient to only enable network priority support at the output ports of the ToR switches that are connected to the end hosts in order to eliminate most queuing delays for grants. However, with more congestion at the network core, every port in the network (including both the ToR and core switches) must be properly configured

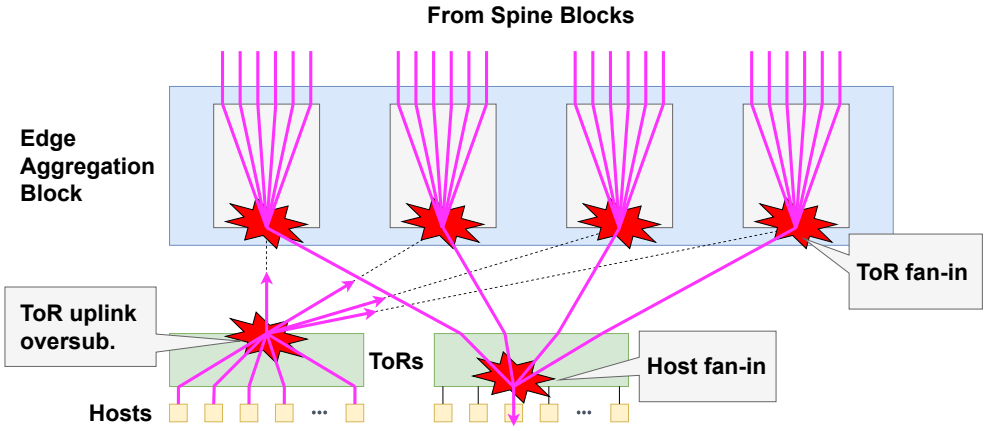


Figure 3.9: Congestion hotspots in a datacenter network (adapted from [89]). In summary, congestion may occur (i) when too many hosts within the same rack are sending to the core (ToR uplink over-subscription), (ii) when more than one hosts are sending to the same host (host fan-in), or (iii) when too many hosts are sending to the same ToR (ToR fan-in). Note that if a congestion is caused by host fan-in, then it's said to occur at the edge of the network; otherwise, it occurs at the core.

to achieve that, which further limits the applicability of the shuffle protocol in a shared network. As a result, in the following discussion, I will start by assuming network priorities are supported only at the edge, then I will discuss the necessity of enabling network priorities also in the core.

In addition to being limited by the core bandwidth, congestion at the core introduces significant new problems to the protocol design, which can cause a further decrease in the shuffle throughput. First, a purely endhost-based shuffle algorithm where each receiver makes its own scheduling decision independently is prone to produce suboptimal bandwidth allocation of the bottleneck links in the core of the network. Second, congestion hotspots can lead to large queues in the network core; as we shall see later, if grants cannot bypass these queues, senders may be constantly starved of grants and under-utilize their uplink bandwidth.

It turns out that the new problems are rather subtle, and the solutions are far from obvious. As a result, the rest of the section is organized as follows. First, I will explain the problems using two concrete examples. Then, I will discuss some (partial) solutions to improve the performance of these examples. Finally, I will introduce a more general approach to analyzing the bandwidth allocation problem in the core and propose a cleaner solution that achieves near-optimal performance but also requires periodic coordination among nodes (thus, it doesn't apply to very low-latency shuffles).

Since the bottleneck is now the bandwidth of the congested core links, it's critical that these links must be fully utilized throughout the shuffle in order to achieve the optimal performance (at the same time, other links must not be under-utilized to a point where they become the new bottlenecks). A

simple way to achieve this goal is (again) via pro rata rate allocation: i.e., messages that traverse the bottleneck core links should always receive bandwidth in proportion to their (remaining) lengths. A nice property of this approach is that all messages can finish at the same time; without this, it's possible that some messages would finish too early, and there are not enough unfinished messages near the end of shuffle to keep the bottleneck links fully utilized.

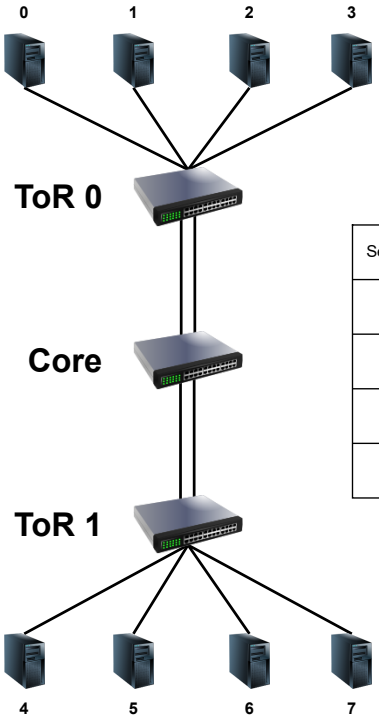
The optimal allocation of the core bandwidth is a new problem that didn't exist previously. When the core bandwidth is abundant and host fan-in is the primary source of congestion, the receivers can enforce the optimal bandwidth allocation of their downlinks via pro rata sliding window sizes. However, now messages will be competing for bandwidth even when they have different receivers. For example, two messages originating from the same ToR will be competing for the uplink bandwidth of their source ToR, and two messages destined to the same ToR for the downlink bandwidth of their destination ToR. In the current scheme, end hosts are making scheduling decisions locally, so they are unable to enforce the optimal bandwidth allocation of the bottleneck links in the core.

Figure 3.10 shows an example. Ideally, the message from node 0 to node 4 should be allocated 3x link capacity compared to the others because it has 3x message size. However, the current scheme will evenly distribute the link capacity among all messages because (a) the sliding window size S is the same for all messages (each receiver only has one incoming message in this example), and (b) the RTT between senders and receivers is also the same for all messages (all data packets experience the same queuing delay at ToR 0). When the throughputs of messages eventually reach the equilibrium, all of them will be equal to $\frac{S}{RTT}$. As a result, the current algorithm will finish 33% slower than the optimal, and Chowdhury et al. [14] shows that the slowdown can be up to 1.5x in more pathological scenarios. Although this is an over-simplified example, this sort of problem occurs repeatedly in the evaluation.

There are at least two solutions to fix the problem in the example above (unfortunately, as we shall see later, neither one can guarantee optimal bandwidth allocation in general). The key idea of both solutions is the same: if a sender or receiver has less data to send or receive, then it should be throttled accordingly to avoid using up too much bandwidth at the congestion hotspots. However, two solutions differ in their mechanisms to achieve this goal.

The first solution introduces packet pacing at the sender side. For example, if the amount of data a sender has to send is only 50% of the maximum outgoing data, then it should also pace itself to transmit at half of the line rate. To implement this solution, all nodes must exchange data among themselves via an all-reduce [85] operation to find out the maximum outgoing data of a node.

The second solution is purely receiver-driven. So far every receiver has the same limit on the total outstanding grants a receiver can have, and this limit only depends on the overcommitment level. To achieve better rate allocation, this solution sets the maximum outstanding grants to be proportional to the amount of data it has to receive (i.e., *pro rata overcommitment*). Thus, in Figure 3.10, node 4 will be able to issue 3x grants compared to any other node, and the sliding window of the message



Shuffle messages

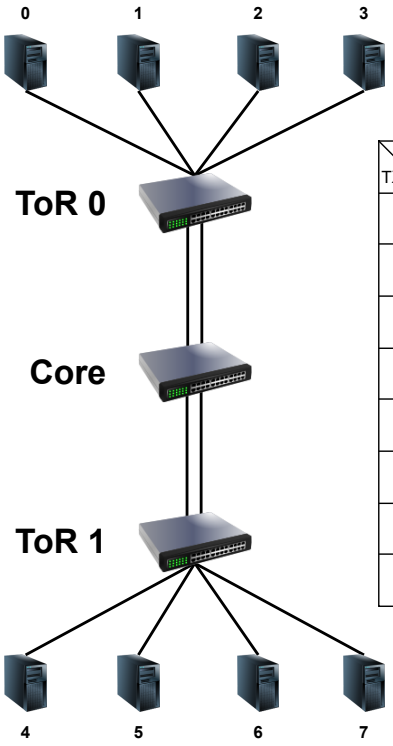
Sender	Receiver	Size (pkts)	Optimal rates (pkts/time unit)	Actual rates (pkts/time unit)
0	4	3000	1	0.5
1	5	1000	0.33	0.5
2	6	1000	0.33	0.5
3	7	1000	0.33	0.5

Figure 3.10: Suboptimal bandwidth allocation of ToR 0's uplink. For simplicity, the diagram omits various complex components in Figure 3.9 and depicts a simple two-level tree topology with 2:1 oversubscription (each end host can send or receive 1 packet per unit of time; the ToR can send or receive 2 packets per unit of time). It is also assumed that packets are perfectly distributed across the core links, so the links between a ToR and the core can be logically considered as one link with 2x capacity. The table shows the messages in the system and their sending rates (optimal vs. actual).

from node 0 to node 4 will also be 3x large compared to any other message. Similarly, this solution also requires an out-of-band communication between all nodes to find out the maximum amount of data a receiver has to receive, then each receiver can reduce its total outstanding grants accordingly.

The second solution is more appealing due to its flexibility and elegance. The biggest problem of sender pacing is that it's too pessimistic and rigid in the use of uplink bandwidth: a sender that has less data to send always ends up under-utilizing its uplink. Though this sender is not the bottleneck now, it could be perturbed in the future and become a straggler. Thus, it's always more desirable to send out the data early while the sender can, and the receivers can decide to throttle the sender if necessary (the bottom line is the receivers are at a better place to make the scheduling decisions). Finally, having two different rate-limiting mechanisms adds complexity to the design and analysis, so it should be avoided if possible.

The second problem introduced by congestion at the core is the increase of queue lengths in the



Shuffle messages

TX \ RX	0	1	2	3	4	5	6	7
0				3.0				4.0
1			3.0				4.0	
2		3.0				4.0		
3	3.0				4.0			
4				4.0				
5			4.0					
6		4.0						
7	4.0							

Figure 3.11: Bandwidth wastage due to grant packets being delayed at congestion hotspots. The network topology is the same as Figure 3.10. The matrix on the right shows the relative sizes of the shuffle messages, where each row represents one sender and each column represents one receiver. For example, if a value of 1.0 is 1000 packets, then a value of 4.0 at position (0, 7) means the size of the message from node 0 to node 7 is 4000 packets.

network. Large packet queues can cause collateral damage to the throughputs of messages from the opposite direction, since their grant packets can be delayed significantly (delay of the data packets is less of a problem because the congested link is already being fully utilized). For example, if there is another message of 3000 packets being sent from node 4 to node 0 in the example of Figure 3.10, then its grant packets will experience significant queuing delay at the uplink of ToR 0. In this example, pro rata overcommitment actually helps mitigate the problem because it reduces the total outstanding grants of node 5, 6, and 7 to one third, effectively reducing the maximum queue length at the uplink of ToR 0.

Unfortunately, pro rata overcommitment doesn't always solve the problem; Figure 3.11 shows a more subtle example. Since all the outgoing data of nodes 4–7 are destined to the first rack, a large queue will be built up at the uplink of ToR 1. Pro rata overcommitment cannot reduce the length of this queue, since nodes 0–3 are the ones with more incoming data; in addition, the total outstanding grants of nodes 4–7 are reduced to $\frac{4}{7}$ of that of nodes 0–3. Thus, the sliding windows

of the messages destined to node 4–7 become too small for nodes 0–3 to fully utilize the uplink of ToR 0. As a result, the cleanest solution to eliminate the collateral damage of large packet queues is to enable in-network priorities in both the ToR and core switches.

The techniques discussed so far in this section already improve the performance effectively when there is limited core bandwidth. However, experiments in Chapter 6 will show that the throughput is still far from the optimal (e.g., roughly 5% and 10% performance loss at median and 90 percentile, respectively), and sub-optimal bandwidth allocation of the congested core links is still the major cause of performance loss even with all the attempts above. This is because these techniques become less effective at larger scales, when there are more than two racks (unlike the examples presented so far).

When the shuffle involves more than two racks, the problem of optimal bandwidth allocation is even more challenging, as there are more paths between nodes in the network. Packets that follow different paths will experience different queuing delays (and have different effective RTTs between their senders and receivers), and messages with smaller (average) RTTs will have an advantage in acquiring larger shares of bandwidth at the congested core links than the competitors. For example, consider two messages m_1 and m_2 of the same size, m_1 is sent from ToR r_0 to ToR r_1, while m_2 is sent from ToR r_0 to ToR r_2; further, assume that only the uplinks of r_0 and the downlinks of r_2 are congested. Both messages will suffer roughly equal delays at the uplinks of r_0, but m_2 will have a larger RTT because its data packets experience additional queuing delay at the downlinks of r_2. Thus, m_2 will receive less bandwidth and finish more slowly than m_1, which leads to under-utilization of r_0's uplinks near the end of shuffle.

Even more interestingly, it turns out that two messages that follow the same path could also suffer from suboptimal bandwidth allocation due to the adaptive sliding window mechanism that is, ironically, designed to mitigate stragglers in the first place. Again, consider two messages m_1 and m_2 of the same size, and they are both sent from ToR r_0 to ToR r_1. Suppose, of all the incoming messages of m_2's receiver, m_2 is the only one whose throughput is reduced at some congested link. Then m_2 will be the only straggler from the receiver's perspective, and the receiver will attempt to increase its sliding window size with the hope that m_2 would be able to catch up faster; thus, the sliding window of m_2 will keep increasing over the course of shuffle. On the other hand, if this is not the case for m_1, then its sliding window will remain roughly constant. As a result, even though the two messages have the same RTT, m_1 will receive much less bandwidth than m_2 because its sliding window is smaller. Ironically, the mechanism used by one receiver to mitigate stragglers among its incoming messages ends up causing a straggler message destined to another receiver. This is another proof that a purely endhost-based protocol doesn't have the correct information to achieve optimal scheduling globally.

One could potentially attempt to design more ad-hoc optimizations to improve the performance for each of the examples above; however, a much better approach is to analyze the problem more

formally under a unified framework. The following paragraphs describe such a simple framework.

First of all, it is well known that the maximum throughput of a message in window-based rate control schemes is given by the following formula (it's also limited by the link speed eventually):

$$throughput(msg(x,y)) = \frac{S(x,y)}{RTT(x,y)} \tag{3.7}$$

where

- $msg(x,y)$ is a message sent from node x to node y,
- $S(x,y)$ is the sliding window size of the message, and
- $RTT(x,y)$ is round-trip time between the sender and receiver of the message, which consists of three components: the network latency of the grant packet, the sender turnaround time (i.e., the time between the sender receives the grant and replies the corresponding data packet), and the latency of the data packet.

This throughput limit will be achieved when the sender has to slow down and wait for grants, which is exactly what happens when multiple messages are contending on some congested link (clearly, the message throughputs are not limited by link speeds in this case).

Second, the vanilla adaptive pro rata rate allocation mechanism (without pro rata overcommitment) computes the sliding window size of message $msg(x,y)$ as follows:

$$S(x,y) = max\_out\_grants \times \frac{msg\_rem\_bytes(x,y)}{total\_rem\_bytes(y)} \tag{3.8}$$

where

- max_out_grants is the maximum number of outstanding grants any receiver can issue, which is a constant that is equal to the product of the degree of overcommitment and $RTTbytes$,
- $msg\_rem\_bytes(x,y)$ is the number of bytes of the message yet to be received by node y, and
- $total\_rem\_bytes(y)$ is the total number of incoming bytes yet to be received by node y.

Said another way, each receiver dynamically distributes its total outstanding grants across all its incoming messages so that their sliding window sizes are always proportional to the remaining bytes (both $msg\_rem\_bytes(x,y)$ and $total\_rem\_bytes(y)$ are changing over time).

Recall that the goal is to achieve pro rata bandwidth allocation of the congested core links, or

$$\frac{throughput(msg(a,b))}{throughput(msg(c,d))} = \frac{msg\_rem\_bytes(a,b)}{msg\_rem\_bytes(c,d)} \tag{3.9}$$

The vanilla algorithm cannot achieve this goal, since Equation 3.8 entails the following instead:

$$\frac{throughput(msg(a,b))}{throughput(msg(c,d))} = \frac{msg\_rem\_bytes(a,b)}{msg\_rem\_bytes(c,d)} \times \frac{total\_rem\_bytes(d)}{total\_rem\_bytes(b)} \times \frac{RTT(c,d)}{RTT(a,b)} \tag{3.10}$$

Pro rata overcommitment can also be evaluated formally under this framework. Recall that the idea was to scale the total outstanding grants in Equation 3.8 in proportion to the amount of total incoming data, so receivers with less incoming data would also maintain fewer outstanding grants:

$$S(x,y) = max\_out\_grants \times \frac{total\_rx\_bytes(y)}{\max_r\{total\_rx\_bytes(r)\}} \times \frac{msg\_rem\_bytes(x,y)}{total\_rem\_bytes(y)} \tag{3.11}$$

where

- $total\_rx\_bytes(y)$ is the total incoming bytes of node y, and
- $\max_r\{total\_rx\_bytes(r)\}$ is the maximum total incoming bytes across all receivers.

When each receiver has only one incoming message (i.e., $msg\_rem\_bytes(x,y) = total\_rem\_bytes(y)$, which is the special case that motivated this approach initially), this approach is a clear improvement over the vanilla algorithm, as longer messages will at least get larger sliding windows. Although this approach turns out to improve performance beyond this special case, it fails to enforce the correct bandwidth allocation in general:

$$\frac{throughput(msg(a,b))}{throughput(msg(c,d))} = \frac{msg\_rem\_bytes(a,b)}{msg\_rem\_bytes(c,d)} \times \frac{\frac{total\_rem\_bytes(d)}{total\_rx\_bytes(d)}}{\frac{total\_rem\_bytes(b)}{total\_rx\_bytes(b)}} \times \frac{RTT(c,d)}{RTT(a,b)} \tag{3.12}$$

An ideal approach would eliminate both $\frac{total\_rem\_bytes(d)}{total\_rem\_bytes(b)}$ and $\frac{RTT(c,d)}{RTT(a,b)}$ from Equation 3.10. Conceptually, there is a simple way to remove the former term: if each receiver is not only aware of its own remaining bytes to receive, but also the maximum remaining bytes across all receivers, then each receiver can use that global information to compute much more accurate sliding window sizes. That is,

$$S(x,y) = max\_out\_grants \times \frac{msg\_rem\_bytes(x,y)}{\max_r\{total\_rem\_bytes(r)\}} \tag{3.13}$$

In practice, it's not possible to access the current value of $\max_r\{total\_rem\_bytes(r)\}$ in Equation 3.13 instantly; one possible workaround is to periodically exchange this information among all nodes via all-reduce operations (so the receivers have to make scheduling decisions based on slightly stale information). As a result, this approach only works for shuffles that run for longer periods of time (i.e., much larger than the latency of the all-reduce operation[1]); otherwise, the errors resulting from stale information won't be negligible.

[1] A hardware-assisted all-reduce operation may complete in 5-10μs for up to 1000 nodes using smart switches [84].

However, eliminating $\frac{RTT(c,d)}{RTT(a,b)}$ from Equation 3.10 turns out to be very difficult, if not impossible. A straw man proposal is to multiply $S(x,y)$ by $RTT(x,y)$, but the resulting sliding windows will be too large to be effective (the actual outstanding grants of $m(x,y)$ will be capped by max_out_grants instead). One could introduce a global constant R to scale back the sliding windows: i.e., multiply $S(x,y)$ by $\frac{RTT(x,y)}{R}$. Ideally, $\sum_{\forall x} S(x,y)$ should be smaller than max_out_grants, but it should not be too small because we still need overcommitment to maintain high link utilization. However, these constraints make it impossible to pick a proper value for R. $\sum_{\forall x} S(x,y) \leq max\_out\_grants$ entails

$$\sum_{\forall x} max\_out\_grants \times \frac{msg\_rem\_bytes(x,y)}{\max_r\{total\_rem\_bytes(r)\}} \times \frac{RTT(x,y)}{R} \leq max\_out\_grants$$

Or, equivalently, $R \geq \sum_{\forall x} \frac{msg\_rem\_bytes(x,y)}{\max_r\{total\_rem\_bytes(r)\}} \times RTT(x,y)$. Note that $RTT(x,y)$ are constantly changing over the shuffle. Thus, if a large R is chosen to account for large RTT's at some receiver, then the sliding windows will be too small when its RTT's become smaller or for receivers that don't experience large RTT's; it is unlikely a single value of R will work for all receivers and at all times.

One could also just leave $\frac{RTT(c,d)}{RTT(a,b)}$ alone. In that case, pro rata rate allocation is only achieved if there is a global uniform RTT. This is obviously not the case in practice: straggler messages almost always suffer from larger queuing delays. As a result, the receiver will constantly underestimate the correct window sizes of the straggler messages, so the stragglers will never be able to catch up.

Fortunately, although not a perfect solution, a heuristic turns out to work rather well in practice. The idea is similar to pro rata overcommitment: scale back the total outstanding grants of a receiver if it has fewer remaining bytes yet to be received than the current slowest receiver (i.e., the one that has the most remaining bytes). That is,

$$S(x,y) = max\_out\_grants \times \frac{msg\_rem\_bytes(x,y)}{\max_r\{total\_rem\_bytes(r)\}} \times \frac{total\_rem\_bytes(y)}{\max_r\{total\_rem\_bytes(r)\}} \quad (3.14)$$

In fact, one could scale back the total outstanding grants even more aggressively by raising the last term above to its n-th power. Experiments show that quadratic scale back ($n = 2$) performs pretty well in practice. Consequently, the formula which describes the bandwidth allocation at steady state ends up with:

$$\frac{throughput(msg(a,b))}{throughput(msg(c,d))} = \frac{msg\_rem\_bytes(a,b)}{msg\_rem\_bytes(c,d)} \times \frac{RTT(c,d)}{RTT(a,b)} \times \left\{\frac{total\_rem\_bytes(b)}{total\_rem\_bytes(d)}\right\}^n \quad (3.15)$$

3.1.8 Putting it all together

Listings 2 and 3 summarize the key parts of the shuffle protocol described so far in python-like pseudocode. For clarity, I left out (i) the code that invoke these methods (e.g., they can be invoked in a busy loop) and (ii) the code that update the global and per-message states upon sending and receiving packets.

```
# Number of packets yet to receive by the slowest receiver in the cluster.
# Updated periodically via some out-of-band communication.
max_rem_rx_pkts = None
# How many more grants is allowed to be issued by this receiver?
grants_left = MAX_OUTSTANDING_GRANTS

class OutMessage:
    """
    Represents an outgoing message at the sender.

    Attributes
    ----------
    total_pkts : int
        Length of the message, in # packets.
    packets_sent : int
        # packets that has been sent out.
    num_grants : int
        How many more packets of this message are permitted to be transmitted.
    """
    # ... class body omitted  ...

class InMessage:
    """
    Represents an incoming message.

    Attributes
    ----------
    total_pkts : int
        Length of the message, in # packets.
    packets_received : int
        # packets that has been received.
    out_grants : int
        # outstanding grants issued to this message by the receiver.
    """
    # ... class body omitted  ...

    def grant_wnd_size(self, rx_pkts_left):
        """
        Compute the maximum outstanding grants that can be issued to this message.

        rx_pkts_left : int
            # packets yet to receive at the receiver of this message.
        """
        msg_rem_pkts = self.total_pkts - self.packets_received
        if max_rem_rx_pkts is None:     # Equation 3.8
            return ceil(max_out_grants * msg_rem_pkts / rx_pkts_left)
        else:                           # Equation 3.14
            return ceil(max_out_grants * msg_rem_pkts * rx_pkts_left / pow(max_rem_rx_pkts, 2))
```

Listing 2: Data structures and global variables

```python
def try_send(out_msgs):
    """
    Find the outgoing message that has the GRPF and grants available,
    and then send one more packet from it.

    out_msgs : list[OutMessage]
        Unfinished outgoing messages
    """
    next_msg = None
    grpf = 0.0
    for m in out_msgs:
        remaining_frac = 1.0 - m.packets_sent / m.total_pkts
        if m.num_grants > 0 and remaining_frac > grpf:
            grpf = remaining_frac
            next_msg = m
    if next_msg:
        next_msg.packets_sent += 1
        next_msg.num_grants -= 1
        send_one_pkt(next_msg)


def try_grant(in_msgs, rx_pkts_left):
    """
    Try to send out one or more grants to senders that are ready to accept
    them. Prioritize incoming messages with larger remaining fractions.

    in_msgs : List[InMessage]
        Unfinished incoming messages.
    rx_pkts_left : int
        # packets yet to receive at this receiver
    """
    while grants_left > 0:
        next_msg = None
        grpf = 0.0
        for m in in_msgs:
            remaining_frac = 1.0 - m.packets_received / m.total_pkts
            if m.out_grants < m.grant_wnd_size(rx_pkts_left) and remaining_frac > grpf:
                grpf = remaining_frac
                next_msg = m
        if next_msg:
            grants_left -= 1
            m.out_grants += 1
            send_one_grant(next_msg)
        else
            break
```

Listing 3: Key scheduling logic of senders and receivers

As mentioned in Section 3.1.2, my algorithm could be viewed as a clean-slate instantiation of the Weighted Shuffle Scheduling (WSS) scheme. Recall that the original paper [14] suggests the use of an external network transport to help achieve weighted fair sharing at the bottlenecks, but it didn't provide a practical transport that is suitable for low-latency shuffles in flash bursts. In contrast, my algorithm provides a satisfying end-to-end solution by exploiting a number of specifities of the problem. For example, the bandwidth-delay product is rather stable in an (unloaded) datacenter network, and it's usually much smaller than the buffer space available at the switches. As a result, each receiver can choose a safe overcommitment level which allows its senders to quickly ramp up their egress rates without causing buffer overflow; there is also no need to employ an adaptive rate discovery scheme like AIMD which may have a slow convergence process. In addition, each node in the datacenter is usually aware of its (maximum) egress bandwidth; thus, a sender can easily saturate its uplink without building up a large queue at the egress port. Finally, in a datacenter environment, the shuffle performance is usually less sensitive to bottlenecks in the network core (see Section 6.3); the performance is already quite good in many cases just by running ensemble-wide scheduling at end hosts using simple ideas such as GRPF and pro rata grant windows. And when limited core bandwidth becomes a problem, a simple heuristic with minimal global coordination is very effective in practice.

Strictly speaking, the GRPF policy may diverge from a weighted fair sharing scheme occasionally. For example, when the receiver of a straggler message finally responds with more grants, a GRPF sender will attempt to allocate all bandwidth to this message for a while before returning to weighted fair sharing. However, this doesn't seem cause any problem in the evaluation.

3.2 Connectionless transport

Starting from this section, I will describe the precise protocol in detail, including aspects that are less essential for performance but nonetheless required by a complete protocol.

The shuffle protocol is connectionless. It can be easily implemented on top of any unreliable datagram transport such as UDP. Each shuffle operation is identified by a globally unique *OPid* generated by the high-level application. To start a shuffle, the application should pass the OPid, the set of nodes (i.e., their network addresses) participating in the shuffle, and the list of outgoing messages to the shuffle module. No setup phase is required, so different nodes may start sending data at different times; this avoids a global barrier which incurs additional latency. Individual incoming messages are returned to the application as they are fully received, and they may complete in any order; further, the packets within each message can be received out of order.

Figure 3.12 summarizes the packet types used in the shuffle protocol. For best performance, small control packets (i.e., GRANT, ACK, and RSEND) should be assigned a higher priority than DATA packets, but it is not mandatory.

DATA	Sent from sender to receiver. Contains a range of bytes within a message, defined by an offset and a length, and the total message length.
GRANT	Sent from receiver to sender. Indicates that the sender may now transmit all bytes in the message up to a given offset.
ACK	Sent from receiver to sender. Indicates that the message has been received in its entirety.
RESEND	Sent from receiver to sender. Indicates that the sender should re-transmit a given range of bytes within a message.

Figure 3.12: The packet types used by the shuffle protocol. All packet types except DATA are sent at a high priority by default.

The application is responsible for managing the shuffle operations. This essentially means three things. First, the application must coordinate among its nodes to generate a unique OPid for each shuffle. This is typically done in an application-specific way: a Spark-like [100] application could use a centralized driver program to generate the OPid, or an SPMD-style [19] application could assign the OPid based on the program structure without further coordination. Second, the application should implement some mechanism to retain incoming packets with unseen OPid's and eventually dispatch them to the correct shuffle operation. Unclaimed packets will expire after some time and be removed from the staging area. Finally, the application must keep track of the completed shuffle operations and use them to filter out stale packets from shuffle operations that are already completed.

The functionality of the shuffle module can be divided into two conceptually orthogonal sub-modules: the sender and the receiver. The top-level shuffle module returns after both sub-modules complete. I discuss these two sub-modules in order below.

3.3 Sender behavior

When a sender starts running, the first task is to divide all the outgoing messages into fixed-size DATA packets. In order to amortize the cost of packet processing, our implementation favors larger packets. For instance, if the underlying network is Ethernet, then all DATA packets except the last one from each message will contain roughly 1.5 KB data (or, up to 9 KB if jumbo frames are used). Note that even if the sender has no data to transmit to a receiver, it should still create an empty DATA packet to this receiver; otherwise, the receiver cannot know the total amount of incoming data.

Recall that each sender should be able to transmit a small amount of data blindly, so the second

task of the sender is to divide each outgoing message into two parts: an initial unscheduled portion, followed by a scheduled portion. For simplicity, it's required that either portion of a message must contain an integral number of DATA packets. To avoid under-utilizing its uplink, the sender sends (slightly more than) one RTT worth of unscheduled packets, so it could receive its first grant before transmitting all the unscheduled data (the number of unscheduled packets must be kept low because having too many unscheduled data reduces the effectiveness of receiver-driven rate control).

As a result, the sender will transmit the first RTTpackets data packets in the sender's GRPF packet schedule unilaterally. However, there is one caveat. If a message doesn't contain any unscheduled data, the sender should still transmit a special DATA packet that contains no message bytes to the receiver unilaterally; the sole purpose of this empty DATA packet is to notify the receiver of the incoming bytes of the message.

As discussed in Section 3.1.4, the sender needs to check the size of its TX queue to decide whether to transmit more packets. When it is time to send the next data packet, the sender simply searches its GRPF packet schedule (starting from the first unsent packet) to find the first outgoing packet that has been granted but not yet sent. Note that only the first RTTpackets entries in the packet schedule are transmissible initially; the sender will update the status of the rest of the packets upon receiving grants from their receivers.

However, there is a small caveat to the above scheme: when the shuffle has uniform data distribution, incoming messages of roughly the same size are likely scheduled to transmit to the receiver at roughly the same time, which creates short bursts of incast. This is because the scheduling priority also indicates the ideal time when the packet will be sent out, given sufficient grants: for instance, if the scheduling priority of a packet is 0.5, then this packet will likely be scheduled to transmit when half of the total outgoing bytes of its sender are finished.

Fortunately, such pathological cases can be easily avoided with randomization: for each outgoing packet, pick a random offset within the packet, and then use the message bytes following that offset (rather than the message bytes starting from this packet) to compute the scheduling priority. For instance, if a message has 10 packets, then the scheduling priority of the 6th packet will be in the range of $(0.4, 0.5]$, as opposed to a fixed value of 0.5 previously. This technique significantly reduces the probability of short incast introduced implicitly by deterministic scheduling priorities.

A sender cannot complete until it's certain that all outgoing messages have been fully received by the receivers. Thus, when a receiver receives a message in its entirety, it needs to reply an ACK to acknowledge the reception of the message; upon receiving the ACK packet, the sender knows it's safe to delete this message. However, in some rare cases, the ACK packet can be lost in the network, which prevents the sender from completion. I discuss the implications of lost packets in Section 3.5.

3.4 Receiver behavior

The main functionality of the receiver is rate control, and, as discussed previously, the receiver enforces two different limits on the outstanding grants. The first limit is a fixed maximum outstanding grants across all senders, and it is used to limit the congestion at the TOR and avoid buffer overflow. The second limit is the per-sender maximum outstanding grants (i.e., the sliding window) that is computed for each incoming message on the fly independently. In the rest of the section, I will first discuss how to compute and enforce these two limits, and then describe how the receiver implements GRPF.

The receiver overcommits its downlink by always maintaining a maximum number of outstanding grants (except when all packets have been granted near the end of shuffle). Thus, unlike the sender, the transmission of grants is event-driven: the receiver can only send out one more grant when the next data packet arrives. The only exception is during the startup when the receiver needs to quickly send out all the grants in batches.

It is straightforward to keep track of the number of outstanding grants except for one tricky case related to unscheduled data. Unscheduled packets can be thought of as grants claimed unilaterally by the senders, so the receiver must increment the counter of outstanding grants by the number of unscheduled packets in a message when its first data packet arrives. As a result, the total outstanding grants can exceed the limit temporarily, but it's fine because the receiver will not send out new grants until the total outstanding grants drop below the limit (as more data packets arrive).

The maximum number of outstanding grants, or max_out_grants, is currently chosen empirically. The experiments suggest that the shuffle performance is usually not very sensitive to the choice of max_out_grants as long as it is not too small. Thus, in order to endure longer periods of interference, max_out_grants can be set to a relatively large value (e.g., 10 RTTpackets); the evaluation will show that buffer occupancy is usually not a concern even with relatively large max_out_grants.

The receiver uses the formula $\texttt{max}\{1.0, \lfloor max\_out\_grants \times \frac{msg\_rem\_bytes}{total\_rem\_bytes} \rfloor\}$ to compute the per-sender maximum outstanding grants for each message on the fly. msg_rem_bytes and $total\_rem\_bytes$ are the remaining bytes yet to be received by the message and the receiver, respectively. Three things are worth noting in the formula. First, the formula ensures that the window size is always greater than zero. Thus, the sum of per-message grant limits can exceed max_out_grants, but this is fine because the total outstanding grants is still limited by max_out_grants. Second, the result of the formula is not a constant over the course of shuffle, as both msg_rem_bytes and $total\_rem\_bytes$ are monotonically decreasing over time. If a message is not making progress, then only $total\_rem\_bytes$ will be decreasing, and thus $\frac{msg\_rem\_bytes}{total\_rem\_bytes}$ will become larger, which means more outstanding grants will be allocated to this sender. Third, the receiver doesn't know the total incoming bytes until it receives the first packets from all senders, so $total\_rem\_bytes$ will not be accurate at the beginning the shuffle. Thus, if the first packets of a few incoming messages arrive earlier than the others, then the receiver could distribute all the outstanding grants across the early messages only according to

the formula. This may not be a big issue in practice, since the receiver will likely hear from all the senders early at the beginning; even if some messages are late, the receiver will immediately reduce the per-sender grant limits of the early messages and prioritize grants to the late ones as soon as they arrive. Or if *max_out_grants* is scaled according to how many total incoming bytes this receiver has relative to the worst-case receiver (Section 3.1.7), then the early messages will only get a small number of grants. Another possible fix is to introduce a slow-start phase which avoids giving out too many grants to a small number of messages early on; I leave this for future work.

Since the receiver doesn't know the sizes of all the incoming messages at startup, it implements GRPF different from the sender. Instead of using a "grant schedule", the receiver keeps a linked list of incoming messages in descending order of their scheduling priorities. The scheduling priority of each incoming message is defined using the same randomized scheme: for each incoming message, randomly pick an offset within the first unreceived packet, and then divide the following unreceived bytes by the total message size to obtain the scheduling priority. When the number of outstanding grants drops below *max_out_grants*, the receiver iterates over the list to find the first message that can be granted one more packet (subject to the per-message grant limit), and then it updates the position of that message in the list accordingly after sending the grant. Using a dynamic linked list makes it easy to add new incoming messages as their first data packets arrive (if a grant schedule is used, it must be recomputed every time a new message is included).

Finally, there is again a small caveat: when the sender and receiver of a message independently select the random offsets, the message will end up having different scheduling priorities on two sides. This is sub-optimal for two reasons. First, if the scheduling priority of a message is higher on the receiver side, then it can increase the turnaround time of a grant, as the sender may not send back the data packet until much later. Second, if the scheduling priority of a message is lower on the receiver side, then the receiver may starve the sender, since the grant is not sent early enough. Admittedly, these two problems are less likely to cause significant performance drop when overcommitment is in use, but, ideally, it's better for both ends of a message to always agree on the scheduling priority. Fortunately, this is rather easy to fix: each sender can pass its random seed to all its receivers along with the first data packets of the messages, and then each receiver can use the random seeds obtained from its senders to generate the same random offset as the sender for each packet when computing priorities (each incoming message uses a different seed).

3.5 Lost packets

The protocol expects packet loss to be rare, so the mechanism for handling lost packets is designed to avoid overhead in the common case where packets are not lost, and remain simple when packets are lost. To this end, I decided to employ a simple lost-packet handling mechanism that is receiver-driven (i.e., receivers are responsible for detecting lost packets) and timeout-based: if a message is

unable to make any progress after a long time (e.g., a millisecond), the receiver sends a RESEND packet that identifies the first range of missing data to the message sender, and then the sender will retransmit the data. This approach is very simple as it handles the loss of DATA and GRANT packets in the same way; and since each receiver already knows the participating nodes in advance, there is no need to treat the loss of the unscheduled packets differently (the receiver cannot detect a lost packet if it is not aware of the incoming message).

There is one special case that is not covered by the receiver-driven timeout mechanism above: the loss of ACK packets. As discussed in Section 3.2, when a message is received in its entirety, the receiver needs to send back an ACK packet to acknowledge the message so the sender can safely exit the shuffle. If that ACK packet is lost, then the sender will get stuck. Senders could use a similar timeout-based mechanism to probe the receivers if the acknowledgements don't arrive in time, but the receivers probably won't be alive to handle these requests because they are free to exit once all data have been received. Thus a better place to handle these requests is the high-level application that keeps track of all the completed shuffle operations: if the designated shuffle is already completed, the application will resend the acknowledgement on its behalf.

Chapter 4

Network Simulator

As part of our design process, I implemented a lightweight network simulator to help us understand the intricate behaviors of different scheduling algorithms, and to iterate faster on research ideas. The entire network simulator contains less than 4000 lines of C++ code, where more than one third of the code is related to collecting various performance metrics. I decided not to reuse existing network simulators [71, 73] that are both more powerful and complex, since it is much easier to instrument and customize a simpler simulator to suit our needs.

There are many advantages in using a network simulator, compared to running experiments on a real cluster, and the most important one is better visibility into the system: it is easy to collect rich low-level performance metrics that would otherwise be impossible. For instance, at any point in time during the shuffle, I can inspect the queue lengths of any switch, count the number of incoming data packets for any receiver, and query how long a packet has been delayed in a queue. The detailed run-time data collected in the simulation can then be aggregated and queried in arbitrary ways to answer questions about the behaviors of the scheduling algorithm. Thus, simulation becomes a really powerful tool that helps us build intuition about the algorithm; in fact, this led directly to the discovery of our window-based rate allocation scheme (Section 3.4).

The second advantage of simulation is easy control over the underlying experiment environment. Recall that our shuffle protocol needs to perform well under various conditions, so it is very useful to be able to change the environment to test our hypothesis quickly during the design. For instance, I often need to vary the size of the cluster, the network topology, the bisection bandwidth available, and introduce different levels of interference. In addition, sometimes in order to get a full picture of the performance, I must run a large number of simulations that exercise all combinations of the factors above; this is clearly non-trivial to achieve with a real cluster.

Finally, using a simulator made it possible to implement oracles or idealized algorithms (e.g., centralized scheduling schemes that require instant access to the global state of the cluster) that cannot be easily done in a real implementation. These oracles help establish theoretical lower bounds

```
while not shuffle_finished:
    # Arrange packets on-the-wire to arrive at their destinations.
    for packet in packets_in_flight:
        if packet.arrival_time == time:
            deliver_packet(packet)
            packets_in_flight.remove(packet)

    # Run the scheduling logic at every node.
    for node in nodes:
        # Attempt to send the next data packet.
        node.try_send()
        # Attempt to issue one or more grants.
        node.try_grant()
        # Attempt to (receive and) process one or more grants and data packets.
        node.try_receive()

    # Run the packet forwarding logic at every switch.
    for switch in switches:
        switch.forward_packets()

    # Advance the simulation timestep.
    time += 1
```

Listing 4: Overall structure of the simulator in Python-like pseudocode.

of the shuffle completion time, and are thus essential to performance evaluation (our goal is to finish in time near to the optimal).

However, like any research that relies on simulation results, the major concern of the simulation approach is credibility, as it is not possible to faithfully model every aspect of the shuffle running in a real environment. To increase the credibility of the research, it would be useful to evaluate a real implementation of the shuffle protocol using a large production cluster, and this is left to future work.

4.1 Overview

In a nutshell, the simulator uses a single-threaded discrete-time architecture: the simulation time is broken up into small fixed-size time slices, or *timesteps*, and the system state is updated according to the events happening in the timestep (the simulation can only progress in the unit of timestep). Thus, the simulator essentially runs a giant loop which advances the simulation one timestep at a time; in each iteration, the simulator moves the packets across the network and executes the scheduling logic for each node. Listing 4 shows the overall structure of the simulator in pseudocode.

In order to simplify the simulator implementation, a timestep is defined to be the time to serialize or deserialize a full data packet by the NIC of a node. This design trades flexibility for simplicity, and it has several important implications on the possible configurations of the simulation. First, the bandwidth of every network link must be a multiple of the bandwidth of the NIC at the end

host, since the switch needs to forward an integral number of data packets per timestep. Second, the latency of one hop in the network must be a multiple of the timestep, since every packet must arrive at the beginning of a timestep. Finally, each message must be consist of a multiple of full data packets, since every message must complete at the end of a timestep. For instance, if one full data packet is 1.5 KB, and the NIC is running at 25 Gbps, then one timestep corresponds to 0.48 μs, and the one-hop network latency can be set to, say, 1.44 μs, or 3x of the timestep. While the configuration of the simulation is somewhat restricted, this design shouldn't lose much generality, since it is always possible to reduce the size of the data packet to run simulations at a finer granularity.

Compared to data packets, control packets such as GRANT and RESEND are much smaller (only a few tens of bytes), so their bandwidth consumption is negligible. For simplicity, the simulation ignores the bandwidth costs of sending and receiving control packets. As a result, although the NIC at the end host can only (de)serialize at most one data packet per timestep, it can (de)serialize any number of control packets on top of that (similarly, a single link can transfer any number of control packets in a single time-step, in addition to one data packet).

However, the bandwidth cost is not the only source of overhead in sending and receiving packets; the software overheads of processing packet are also important. For instance, after the NIC receives a data packet from the network and puts it in a queue in the host memory, the shuffle application may need to assemble the message by copying the bytes out of the packet.

Unfortunately, software overheads are very hard to model accurately in simulation, since they are heavily influenced by the memory bandwidth and cache miss latency (the former usually dominates the cost of processing data packets, while the latter dominates the cost of processing control packets). For simplicity, the simulator currently assumes that the costs of processing data packets are much smaller than the bandwidth cost, and it ignores the costs of processing control packets (the receiver can fetch incoming packets in batches to amortize the cost of cache misses); Section 4.4 will discuss the limitations of this approach.

As a result, the receiver can process incoming packets faster than they are deserialized; the exact number of data packets that can be processed per timestep is configurable in the simulation, and the receiver is free to process any number of grants per timestep. Figure 4.1 shows an example where a receiver has just been rescheduled after preemption, and it's trying to catch up on the packets that have built up for it in memory during the preemption.

It is also rather easy to simulate the two types of interference that are considered in this work: node preemption and link bandwidth variation. When a node is preempted, the simulator simply doesn't run the per-node scheduling logic (i.e., line 10-15 in Listing 4) for this node. Although the receiver will not be able to process any incoming packets during this time, the NIC will continue to receive and place them in the RX queue. Bandwidth variation of a link can be simulated by adjusting the number of data packets that can be moved across the link per timestep. For example, if the uplink bandwidth of a node is reduced to 30%, then the simulator can maintain a counter for

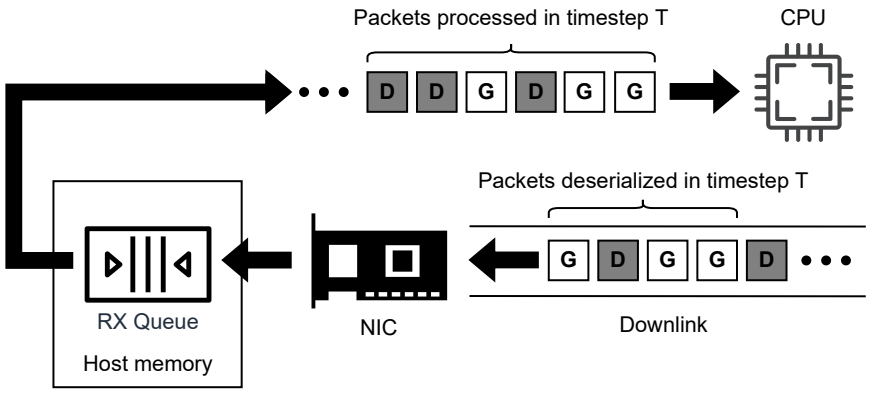


Figure 4.1: Flow of incoming packets from the downlink to the CPU. Packets can be processed faster than they are deserialized: e.g., each receiver is configured to process at most 3 data packets plus any number of grants per timestep.

this link which increments by 0.3 per timestep; when the counter is greater than or equal to 1.0, the simulator can move one data packet across the link and subtract one from the counter.

The two major tasks of the simulator are (i) scheduling messages at the end hosts and (ii) moving packets across the network. The following two sections discuss these two tasks respectively.

4.2 Message scheduling

Scheduling algorithms are rather easy to implement in the simulator. To implement a new algorithm, one just need to override the *node.try_send*() and *node.try_grant*() in Listing 4 with its new sender-side logic and receiver-side logic, respectively. Each algorithm usually takes only a few tens of lines of code; further, implementing centralized scheduling algorithms is equally easy, since the simulator has access to the global state.

The software overheads of running the scheduling algorithm are again ignored in the simulation. Thus, the scheduling decisions can be made instantaneously at the beginning of each timestep, and then the outgoing packets are passed to the NIC for serialization. Similar to the software overheads of processing packets, in reality, the scheduling costs are not negligible compared to the bandwidth costs; in fact, the software costs will eventually become the bottleneck of shuffle, since the CPU will not be able to keep up with ever-increasing network speeds. Section 4.4 will discuss the drawback of ignoring software costs in more detail.

4.3 Moving packets

There are only two tasks that need to be done in order to simulate moving packets across the network: (i) scheduling outgoing packets to arrive at the next hops sometime in the future, and (ii)

forwarding incoming packets that arrive at the switches to the correct output ports.

The first task is implemented using a list that keeps track of all the packets currently on any wire, sorted in increasing order of their scheduled arrival times. When an outgoing packet is about to be sent across a link, its scheduled arrival time at the next hop will be set to the current time plus the one-hop link latency. At the beginning of each timestep, the simulator removes packets that are scheduled to arrive at current time from the list, and delivers them to the destinations (i.e., line 3-6 in Listing 4); the destination of a packet is always an RX queue at a switch or end host, where the newly arrived packets are simply appended. The process above applies to both the data packets and control packets.

The second task requires every switch to forward all the arriving packets at the current timestep from its input ports to output ports. While each input port only has one RX queue, the number of TX queues at an output port depends on the number of network priorities supported by the switch (usually up to 8). However, our shuffle protocol uses at most two network priorities: a high priority for the control packets, and a low priority for the data packets. Thus, the control packets will always be forwarded to a TX queue at a higher priority than the data packets, which allows them to be transmitted instantaneously before any data packet. For simplicity, the simulator assumes infinite buffer space, so it never drops packets due to buffer overflow.

4.4 Limitations

The major concern of relying on a simulation approach is that the behavior of the simulator can diverge from reality, which undermines the credibility of the results. Unfortunately, it is very difficult, if not impossible, to accurately model the behavior of the shuffle protocol in a simulator. This is due to at least two reasons. First, the conditions of the underlying environment are often changing in a complex and irregular way in practice depending on the workloads: e.g., studies [104, 44] show that the network traffic patterns of today's datacenter workloads are often bursty over very short time intervals. Second, different components in the system interact with each other in a non-deterministic way, and they may even compete with each other for shared resources such as CPU time, memory, and caches. For example, the sender and receiver modules may be running on different threads that are subject to the thread scheduler of the underlying OS. Further, in order to saturate the network bandwidth, CPU-intensive tasks such as sending and receiving packets are usually spread across multiple cores, and then constantly load-balanced; this can easily create contention on shared memory and slow down the system. Thus, to avoid over-complicating the simulator, I had to make some approximations.

First, the simulator currently ignores all software overheads, since they are notoriously hard to model in simulation: in reality, the execution time of a task is often heavily influenced by non-deterministic factors such as memory access latency and cache coherency overhead. There are two

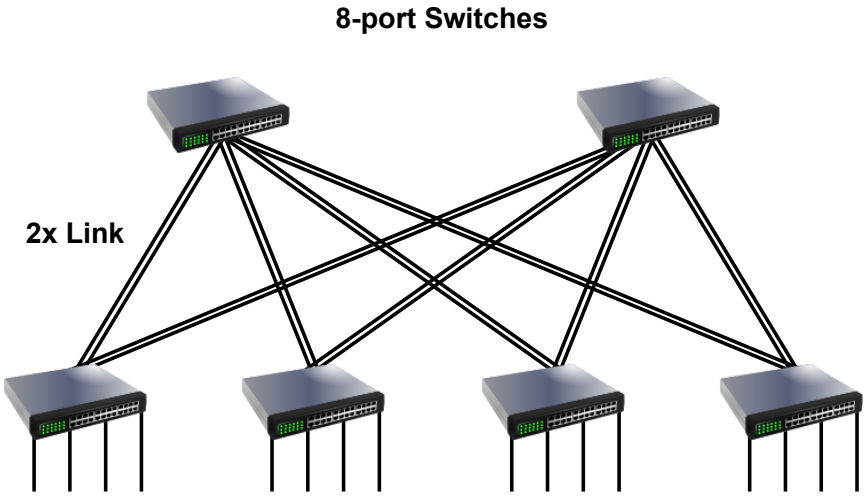


Figure 4.2: A two-level fat tree topology built from 8-port switches.

major sources of software overhead in a shuffle: one is finding the next message to transmit or grant, and the other is sending and receiving network packets. While our scheduling policy might be simple enough to be implemented efficiently on multi-core machines, the packet processing overheads have been a well-known bottleneck for software-based transport protocols [76]; even highly-optimized userspace protocols such as Snap [65] suffer from this problem and have to burn a significant number of cores just to saturate the network bandwidth. The simulator currently assumes network bandwidth is the only limiting factor of shuffle performance; if software overheads turn out to be the bottleneck, then the shuffle protocol probably needs to be implemented in a smart NIC to keep up with the network speed.

Second, the simulator doesn't accurately model the network topology that is being used in reality. The current simulator implements an idealized multi-level fat tree topology where each intermediate switch is connected to exactly one higher-level switch (Figure 3.1); switches that are nearer the top of the hierarchy also have higher aggregate bandwidth (or, equivalently, a larger number of ports). However, this setup is not feasible in practice as commodity switches only support a small fixed number of ports. Figure 4.2 shows a realistic two-level fat tree topology. Since each switch can now connect to more than one switches in the next level, additional load-balancing techniques, such as randomized packet spray [22], must be employed to distribute packets evenly across the available paths. These complex behaviors are obviously beyond the scope of the current simulator; in order to model them, one must resort to more advanced network simulators such as [71, 73].

Third, the interference in the real world will likely be quite different from our simulation. Thus, without actually deploying the shuffle protocol in a production environment, it is hard to tell how faithful the simulation results are.

The best way to work around the limitations of the simulator is to conduct the experiments using a real implementation, and I leave this to future work.

Chapter 5

Workloads

One important design goal of the shuffle protocol is that it should work well for all possible shuffle patterns. Unfortunately, prior works usually omit this problem, and the shuffle workloads used in evaluating performance are overly restricted. As a result, I had to design new workload generators to conduct the evaluation. This chapter discusses the challenges in workload generation, the solutions I arrived at, and their limitations.

As discussed in Section 3.1.1, shuffles that have uniform data distribution (i.e., every node sends and receives the same amount of data) are more challenging than others, as they require the link bandwidths of all nodes to be fully utilized simultaneously in order to minimize the shuffle time. Thus, the following discussion focuses on shuffles with uniform data distribution; however, the workload generators presented in this chapter can easily produce workloads with skewed data distribution as well.

Every shuffle can be specified with an $N \times N$ *shuffle matrix*, where N is the number of nodes: the element s_{ij} at position (i, j) denotes the size of the message sent from the i-th node to the j-th node. In addition, this matrix can be *normalized* by dividing each element in the matrix by the average message size: i.e., in the resulting matrix, the sum of all elements is equal to N^2. Thus, for shuffles with uniform data distribution, each row or column in the normalized shuffle matrix must add up to N.

The workload generators always output normalized shuffle matrices, since they made it possible to compare shuffle workloads that differ in data sizes. To instantiate a concrete shuffle workload, the simulator specifies the average message size and computes the actual message sizes based on that. For simplicity, the term shuffle matrix will simply refer to the normalized matrix from now on.

Informally speaking, to better exercise the protocol and get a full picture of its performance, the workload generator needs to produce a variety of shuffle patterns that are "different" from each other. However, there is no canonical way to fully characterize shuffle patterns. In this thesis, I decided to focus on one particularly important aspect, i.e., the variety of message sizes, or *message skewness*,

	0	1	2	3	4	5	6	7	8	9
0	0.52									9.48
1	0.66								8.82	0.52
2	0.46							8.36	1.18	
3	0.40						7.96	1.64		
4	0.46					7.50	2.04			
5	0.62				6.88	2.50				
6	0.56			6.32	3.12					
7	0.56		5.76	3.68						
8	0.60	5.16	4.24							
9	5.16	4.84								

(a)

	0	1	2	3	4	5	6	7	8	9
0							0.47	9.53		
1								0.30	9.70	
2							0.03		0.15	9.82
3	0.41		0.43	0.02		9.14				
4	1.15		0.41	7.45		0.39	0.48	0.02	0.02	0.08
5	1.18	0.02	0.72	0.91	0.25	0.38	6.32	0.10	0.09	0.03
6	0.76	0.77	0.74	0.56	6.28		0.78	0.03	0.04	0.04
7	2.93	5.44	0.75	0.03	0.09		0.76			
8	3.06	0.42	5.25	0.41	0.41		0.40	0.02		0.03
9	0.51	3.35	1.70	0.62	2.97	0.09	0.76			

(b)

Figure 5.1: Two matrices with the same message skewness of 0.75. They are generated with the two approaches in this chapter, respectively. Even though the message skewness is the same, the matrix on the left is less challenging for many shuffle algorithms, since every sender only has a few non-zero messages to choose from.

since it has a profound influence on the protocol design as discussed in Chapter 3. Section 5.3 will discuss the limitations of this approach.

A simple way to measure the message skewness is to use the standard deviation of the (normalized) message sizes. Note that for any shuffle that has uniform data distribution, the maximum possible standard deviation of its message sizes depends only on N, the number of nodes in shuffle, and it can be obtained when there is exactly one value of N in each row and each column (i.e., each node sends its entire data to a distinct node): i.e., $max\_stddev = \sqrt{\frac{N\times(N-1)^2+(N^2-N)\times(0-1)^2}{N^2}} = \sqrt{N-1}$. It is less convenient to present the experiment results when the possible range of message skewness increases with the number of nodes.

Thus, the message skewness is actually defined as the *normalized standard deviation*, that is the standard deviation of the message sizes divided by the maximum possible standard deviation, so its value is always within the range of $[0, 1]$ regardless of the cluster size.

As a basic requirement, the workload generators should produce shuffles that cover the entire spectrum of message skewness. In addition, they should also attempt to generate "interesting" cases that are beyond the scope of message skewness: e.g., Figure 5.1 shows two shuffle matrices that are very different but happen to have the same message skewness. However, this problem turns out to be non-trivial, and I have yet to find a single satisfying solution. As a result, I ended up implementing two workload generators, each with its own pros and cons. The following two sections discuss these

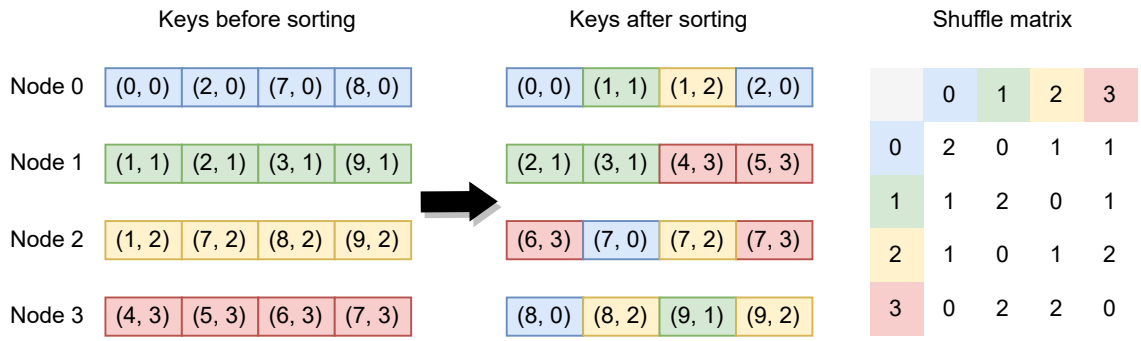


Figure 5.2: Distributed bucket sort and its corresponding shuffle matrix. Each input key is denoted by a pair $(key, node\_id)$, where the extra $node\_id$ tag specifies the original node where key resides; $node\_id$'s are used to break ties when keys are equal, which maintains the sort stability.

two approaches, respectively, and Section 5.3 discusses their limitations.

5.1 Sort-based workload generator

The first workload generator is inspired by the experiment of MilliSort: each distributed bucket sort instance needs to perform a final shuffle step to transmit the input keys to their final destinations, so there is a one to one relation between distributed bucket sorts and shuffle workloads. As a result, to generate an $N \times N$ shuffle matrix, the generator works as follows (unlike distributed sorts, the generator works locally):

1. Randomly generate M initial keys for each of the N nodes.
2. Compute the partition of the key range so that each node ends up hosting exactly M keys.
3. Count the sizes of the shuffle messages (i.e., the number of keys sent between nodes).
4. Fill the shuffle matrix with the message sizes (normalized to make sure that each row or column adds up to N).

Further, M should be much larger than N so that more different message sizes can be generated; in practice, $M = 100 \times N$ appears to be sufficient. Figure 5.2 shows a distributed bucket sort and its resulting shuffle matrix.

With this approach, it is easy to ensure that the resulting shuffle workloads always have uniform data distribution, since each node ends up hosting the same number of keys. In addition, experiments showed that, by tweaking the random distribution of the input keys, it is possible to produce shuffle workloads with varying degrees of message skewness (discussed below). However, it turned out that

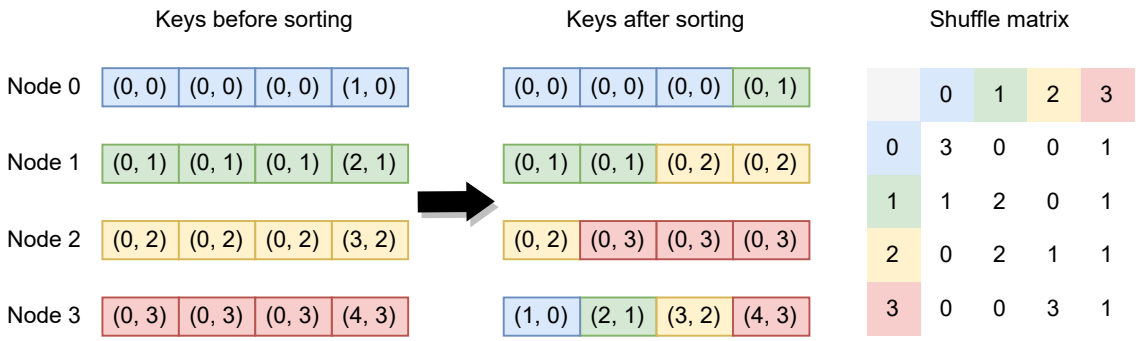


Figure 5.3: Message skewness resulting from duplicate keys.

the message size distribution is affected by the initial key distribution in a rather contrived way, and the resulting shuffle matrices are limited to very specific patterns.

The fact that the resulting message skewness can depend on the key distribution may be counter-intuitive at first: given a fixed key range, the average number of random input keys that fall within that range should be equal for every node, regardless of the distribution used to generate the keys. Thus, it appears that the resulting shuffle workloads should always have uniform message distribution. However, this is only true if duplicate keys are rare (e.g., the keys are real numbers) and the number of keys per node, M, is sufficiently large.

Figure 5.3 shows a simple example where a majority of the input keys are equal. In this particular shuffle matrix, non-zero values are concentrated along the major diagonal and the last column. In reality, this pattern can occur when the input keys are drawn from a heavy-head Zipfian distribution.

One caveat from the example above is that, strictly speaking, uniform data distribution doesn't mean that each node sends and receives the same amount of data (since nodes may keep different amounts of data to themselves). If a majority of the data remain local during the shuffle (e.g., most non-zero values in the matrix occur on the major diagonal), then it is not a very interesting test case for the purpose of measuring shuffle performance. Thus, in practice, a post-processing step is added to randomly shuffle the rows and columns of the matrix, which prevents such pathological cases.

Changing the input key distribution also changes the patterns of the shuffle matrices. For example, if a highly skewed discrete normal distribution is used to generate the input keys instead, the patterns will be significantly different. Further, one can even mix and match multiple distributions to increase the variety of patterns: i.e., to generate a key, first select a distribution randomly, and then draw from that distribution.

As a result, the message skewness produced by this approach is almost a by-product of duplicate keys, and it can only generate shuffle matrices that have very specific patterns. While these specific patterns are valuable corner cases in evaluating performance, it's questionable whether they provide a full picture of the performance faithfully.

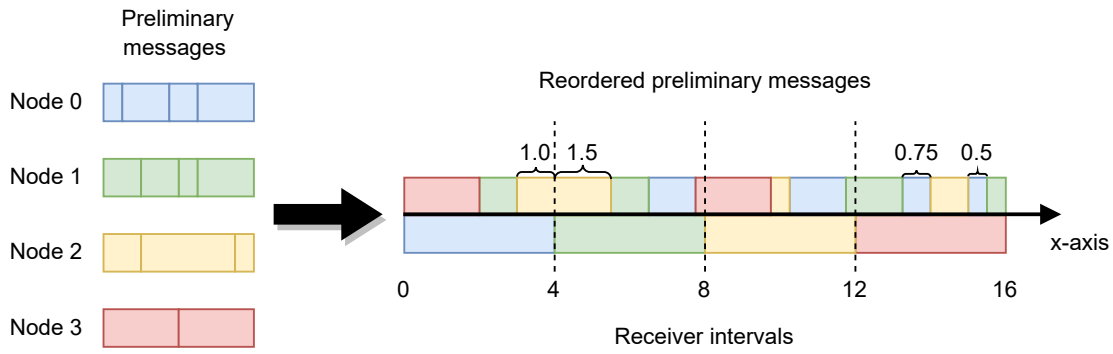


Figure 5.4: An algorithm to enforce column sums of N. The largest yellow preliminary message crosses the boundary of the first two receiver intervals, so it is split in the resulting matrix: i.e., Node 2 will send 1.0 and 1.5 units of data to Node 0 and 1, respectively. Further, two blue preliminary messages fall within the last receiver interval together, so they are merged into one final message in the resulting matrix: i.e., Node 0 will send a total of 1.25 units of data to Node 3.

Despite being limited to specific patterns, this approach can actually generate shuffle workloads that span the entire spectrum of the message skewness factor; unfortunately, this simply means that the message skewness cannot capture all the interesting characteristics of shuffle workloads.

5.2 A more general approach

The biggest drawback of the first approach is the lack of direct control over the message size distribution (the message skewness is merely a side-effect of duplicate keys). Thus, a natural idea is, why not generate shuffle messages directly from a given random distribution?

With this idea, the process of workload generation can be divided into two steps: (i) generate a random distribution of message sizes, and (ii) generate the shuffle matrix by sampling from that distribution. Step one is rather straightforward, so I will discuss it later at the end of the section. However, there is a key challenge in step two: to generate shuffles with uniform data distribution, every row or column in the shuffle matrix must add up to N. While it is rather easy to enforce either the row sums or the column sums, the challenge is how to enforce both of them simultaneously.

For example, to satisfy the requirement on the row sums, the shuffle matrix can be generated one row at a time as follows. For each row, the generator keeps track of the cumulative sum of the generated elements. If the current cumulative sum exceeds N, truncate the last generated element and set the rest of row to 0. Or, if all elements in a row are generated and the sum is smaller than N, increase the last element by the delta between the current sum and N. Note that there are many other ways to enforce the row sums; another possible approach is to generate all elements in a row first, and then scale the elements proportionally to adjust the row sum.

However, the shuffle matrix resulting from the above process cannot guarantee that every column

adds up to N. Fortunately, motivated by the sort-based workload generator in Section 5.1, there is a simple algorithm that adjusts the matrix elements to satisfy the column sum requirement without changing the row sums. Thus, the input matrix of the algorithm is referred to as the *preliminary* shuffle matrix, and its elements denote the preliminary message sizes.

To help explain the algorithm, imagine that the total amount of data to be received by a node is represented by a line segment of length N, or a *receiver interval*, and that each preliminary message of size s is represented by a line segment of length s. Given a preliminary matrix m where the row sums are always N, the algorithm reconstructs a new matrix m' where the column sums are also N as follows:

1. Take all of the preliminary messages (each labeled with its sender) and sort them randomly into a list.

2. Arrange the lists of preliminary messages and receiver intervals side by side.

3. Walk through the preliminary messages in order, assigning bytes to receivers based on how many bytes have fallen within each receiver interval. Preliminary messages that span receivers get split in two final messages, one for each receiver.

Figure 5.4 illustrates an example run of the algorithm. Note that each outgoing preliminary message can overlap with at most two receiver intervals, and multiple preliminary messages from one sender can overlap with the same receiver interval.

It is easy to see that the row sums in the new matrix remain unchanged, since the process above can only change the destination node of a preliminary message, but not its source node (so the total outgoing data of each node is still N).

One problem of the algorithm above is that the message size distribution may be distorted in the final matrix: while the messages in m are generated from the given distribution, the message sizes finalized in m' could be quite different as the preliminary messages are split and/or merged in the process. Ideally, every preliminary message resides in a distinct receiver interval, so m' will contain the exact same elements as m. However, this is usually not possible.

Fortunately, there is a best-effort greedy algorithm that turns out to be very effective in reducing the distortion of the message size distribution in practice. This algorithm is inspired by the following key idea. Suppose the values in each row of the preliminary matrix can be rearranged so that every column also adds up to roughly N. Then traversing the resulting matrix in a column-major order will produce a placement of preliminary messages that has the following two nice properties, when aligned with the receiver intervals. First, there will be very little crossover with the receiver intervals, since column sums are roughly N. Second, it is extremely unlikely that two preliminary messages from the same sender will fall within the same receiver interval, since there are always $N - 1$ preliminary messages (including empty ones) from other senders in between.

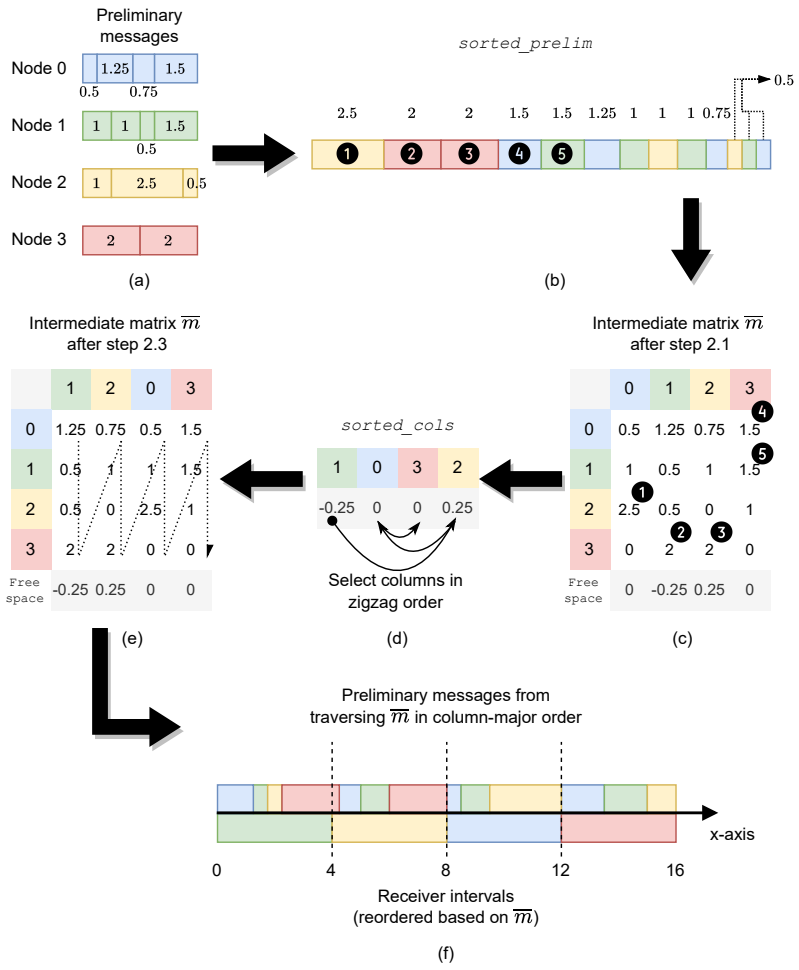


Figure 5.5: Greedy algorithm that optimizes the placement of preliminary messages. The preliminary messages are the same as Figure 5.4, but only one crosses the boundary this time.

More precisely, this greedy algorithm replaces the random shuffle in step 1 above, and decides the order of preliminary messages in a smarter way as follows (ties are always broken randomly below):

1. Sort all preliminary messages in descending order of lengths; store the results in *sorted_prelim*.
2. Construct an intermediate matrix $\bar{m}$ where each row contains the same set of elements as the preliminary matrix m, but in a potentially different order:
   2.1. Decide how to place each preliminary message in the order of *sorted_prelim*. If a preliminary message is sent by the i-th node, then it will stay in the i-th row of $\bar{m}$; further, it will be placed in the column that currently has the maximum *free space* (i.e., N minus the sum of elements already placed in that column), in order to avoid overcommitting

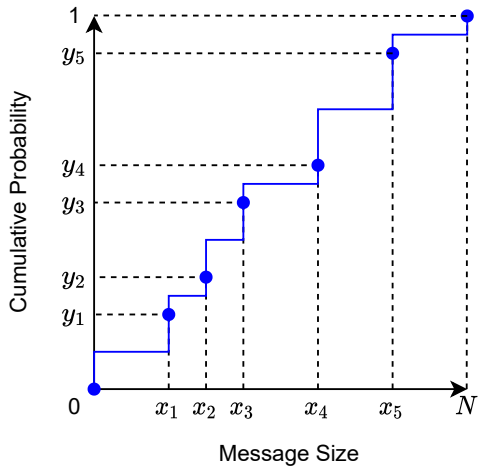


Figure 5.6: A generated cumulative distributed function (CDF) of message sizes.

columns too much (when a column is over-committed, its free space will be negative).

2.2. Once all preliminary messages are placed in $\bar{m}$, sort the columns in ascending order of their free space, and store the results in *sorted_cols* (i.e., overcommitted columns appear before under-committed ones) to be used in the next step.

2.3. As an additional optimization to reduce crossovers, rearrange the order of columns in $\bar{m}$ to place over- and under-committed columns next to each other, with the hope that their negative and positive free spaces would cancel out each other: i.e., when finalizing the order of columns from left to right, always select either the head or tail of *sorted_cols*, whichever minimizes the *absolute value* of the cumulative column free space (the chosen column is removed from *sorted_cols* afterwards).

3. Output the final placement of preliminary messages by traversing $\bar{m}$ in column-major order; in addition, rearrange the receiver intervals to match the order of columns in $\bar{m}$.

Figure 5.5 illustrates the greedy algorithm using the same example as Figure 5.4.

So far, I have shown how to compute shuffle matrices, given a random distribution of message sizes. Now the only problem left is how to create random distributions that cover a variety of means and standard deviations. Since the row sums of the shuffle matrix are N, each distribution can be represented as a CDF where the maximum x value is also N. Thus, a CDF of the message size can be generated in two steps:

1. Select n values uniformly at random on both the x-axis and y-axis, x_i and y_i ($1 \leq i \leq n$), then sort the values so that $0 < x_1 < x_2 < ... < x_n < N$ and $0 < y_1 < y_2 < ... < y_n < 1$.

2. Connect the following points, $(0,0),(x_1,y_1),(x_2,y_2),...,(x_n,y_n),(N,1.0)$, sequentially using a stepped line (or some curve) to finish the CDF.

Figure 5.6 shows an example CDF generated with this approach. There are certainly other ways to generate the CDF of message size, and I have no concrete evidence whether this approach is better (or worse) than others; I chose this approach because it's simple and proves to be effective.

In practice, this algorithm is run repeatedly with varying values of n, and the generated CDFs are classified into different buckets based on the means and standard deviations as follows: (i) divide the ranges of means and standard deviations into fixed-size intervals, and (ii) assign each bucket an interval of means and an interval of standard deviations. This process terminates once every bucket contains at least some minimum number of CDFs. Figure 5.7 shows a number of example random CDFs output by this process.

Our experiments show that this more sophisticated approach can also cover the entire spectrum of the message skewness. Compared to the sort-based workload generator, this workload generator has much better control over the message size distribution, and it can generate many shuffle patterns that are not possible with the former approach. However, the sort-based workload generator is still valuable, since it can generate corner test cases where different rows/columns have vastly different message skewness. Such workloads are highly unlikely to be generated with the latter approach, since it always samples message sizes from the same CDF for each shuffle matrix; even if it uses a different CDF per row, the probability to generate those corner cases is still extremely low.

5.3 Limitations

The biggest concern of the current workload generators is that the notion of message skewness cannot fully capture the characteristics of shuffles. For instance, even though both generators presented in this chapter can cover the entire spectrum of the message skewness, the workloads generated from them are visibly different even when they have the same message skewness. As a result, before inventing any new workload generators, it is crucial to come up with new metrics that can capture the other interesting aspects of shuffles; otherwise, it is not even possible to properly evaluate the expressiveness of any new workload generators.

One other possible interesting aspect of shuffles is the degree of node connectivity, or how many (non-zero) messages a node needs to send or receive. For instance, for shuffles used in data analytics (e.g., distributed sorts and joins), every node will likely need to communicate with every other node. In contrary, shuffles that appear in fluid simulations probably have much lower connectivity, since each node only needs to exchange data with other nodes that hold its neighboring data partitions.

In addition, the irregularity in connectivity patterns could be interesting as well. For example, in graph analytics, if the degrees of nodes vary dramatically (e.g., in social graphs), then the resulting shuffles will have highly irregular connectivity patterns (partitions that contain "hotspot" nodes will

likely send more messages and/or data than others). Although the sort-based workload generator can already produce matrices with irregular rows and columns, this approach is far from satisfying as mentioned before.

It is not yet clear how many other interesting aspects of shuffles exist, whether they will impact the current protocol design, or how to design new workload generators to cover them. They are left for future work.

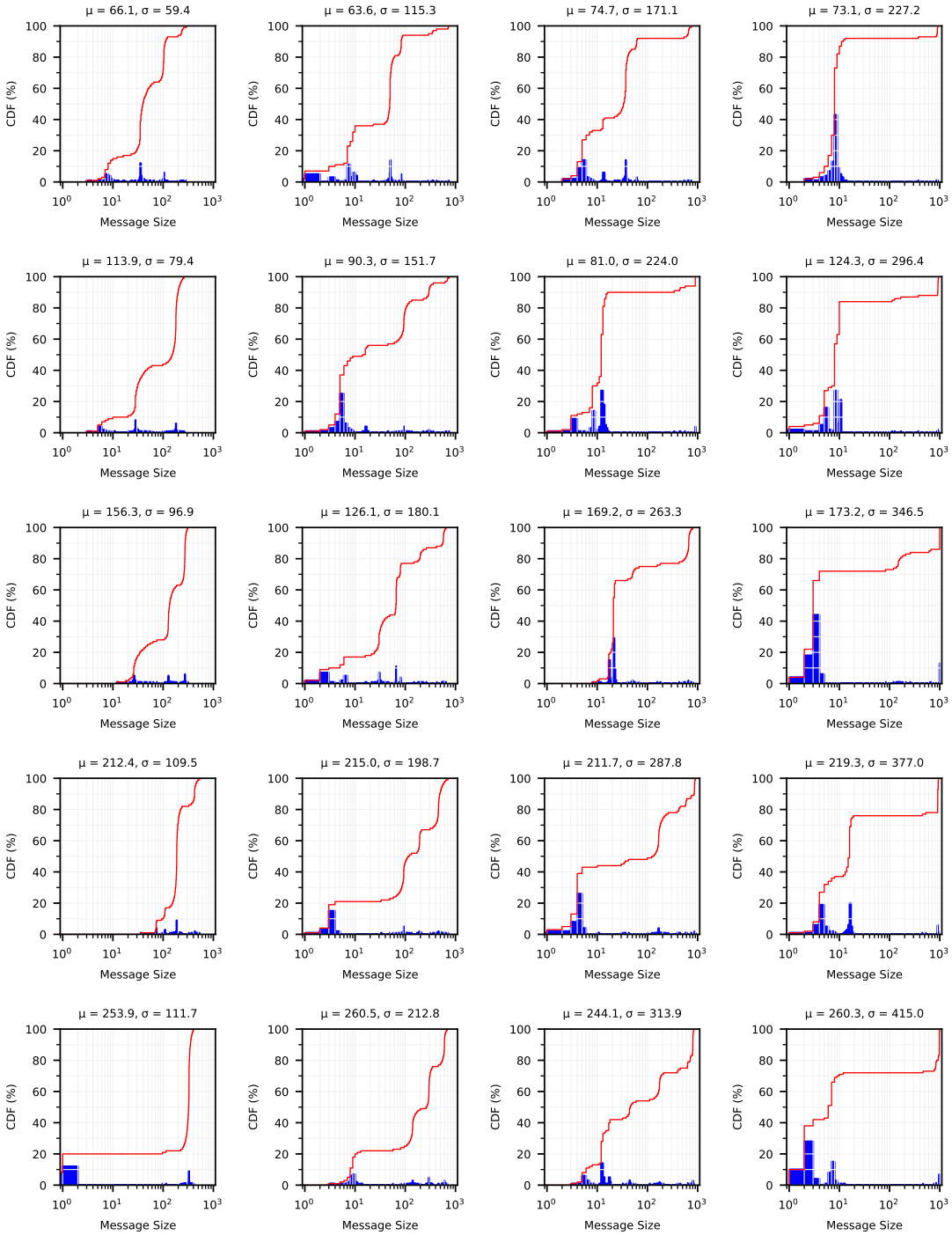


Figure 5.7: Some example random CDFs of messages sizes ordered by their means and standard deviations (i.e., μ and σ). $N = 10^3$. μ increases from top to down in each column, and σ increases from left to right in each row. In each graph, the red curve denotes the cumulative probability, and the blue shaded areas denote the probability density function. The x-axis is log-scale.

Chapter 6

Evaluation

This chapter evaluates the shuffle protocol described in Chapter 3 via network simulations. From now on, this protocol will be referred to as the GRPF algorithm, for lack of a better name [1]. My goal was to answer the following questions:

- Does the GRPF algorithm provide near-optimal performance for all kinds of workloads and under various conditions?
- How does the GRPF algorithm compare to existing approaches?
- How important are the various mechanisms of the GRPF algorithm to its performance?

The following configurations are fixed for all simulations in this chapter. The link speed of each node is 25 Gbps. A full data packet is 1500 bytes, so a timestep in simulation is roughly 0.5μs. Within a single rack, the round-trip time between nodes is 8 timesteps (or 4μs); if two nodes are in different racks, then the round-trip time increases to 16 timesteps (or 8μs). Finally, the cluster has an idealized two-level fat tree topology where each rack contains a total of 40 nodes (the over-subscription factor varies depending on the experiments).

Also note that the workloads used in all simulations are generated with the more general approach in Section 5.2; the results with the sort-based workload generator (Section 5.1) are more or less similar and thus omitted from this chapter.

The evaluation will focus on three algorithms. The first one is the GRPF algorithm, which is the major contribution of this dissertation; it uses an overcommitment level of 10 by default. The other two are the Hadoop and MADD algorithms; they are implemented (with slight adaptation) in the simulator based on the descriptions in Chowdhury et al. [14, 15]. I did not include the WSS algorithm because this would require simulating a full-fledged network transport to be used by WSS. Lockstep-style algorithms are too fragile and thus omitted here.

[1] Note that GRPF is a local policy used in the ensemble-wide scheduling of end hosts; it's just a small piece of the complete protocol.

The primary goal of the Hadoop algorithm is to achieve fair bandwidth sharing; in addition, it limits the maximum number of concurrent connections a receiver can open to prevent large incast. A common way to implement this algorithm is to have each receiver open multiple TCP connections to random senders to receive data from, then the receivers can rely on TCP to achieve fair sharing among the incoming flows. However, this approach requires a faithful implementation of TCP, which is too complicated and beyond the scope of this work. Thus, I decided to implement the algorithm in a simpler receiver-driven scheme: each receiver randomly selects a fixed number of messages to grant (it will not grant to other messages until one of these messages is fully granted), and it employs a constant sliding window size of RTTpackets per message (unlike the GRPF algorithm, there is no need to enforce a limit on the maximum outstanding grants per receiver separately). On the sender-side, a round-robin policy is used to share uplink bandwidth evenly across outgoing messages: when it is time to send the next data packet, the sender will choose the least recently transmitted message that have grants available. The result is an implementation that achieves the max-min fair sharing policy (for the same reason why pro rata sliding windows implement weighted max-min fair sharing (c.f. Theorem 3.1.1); I believe this adaptation of the original TCP-based Hadoop algorithm will provide a more meaningful assessment of the fair-sharing scheme. For simplicity, I will still refer to it as the Hadoop algorithm in the evaluation[1].

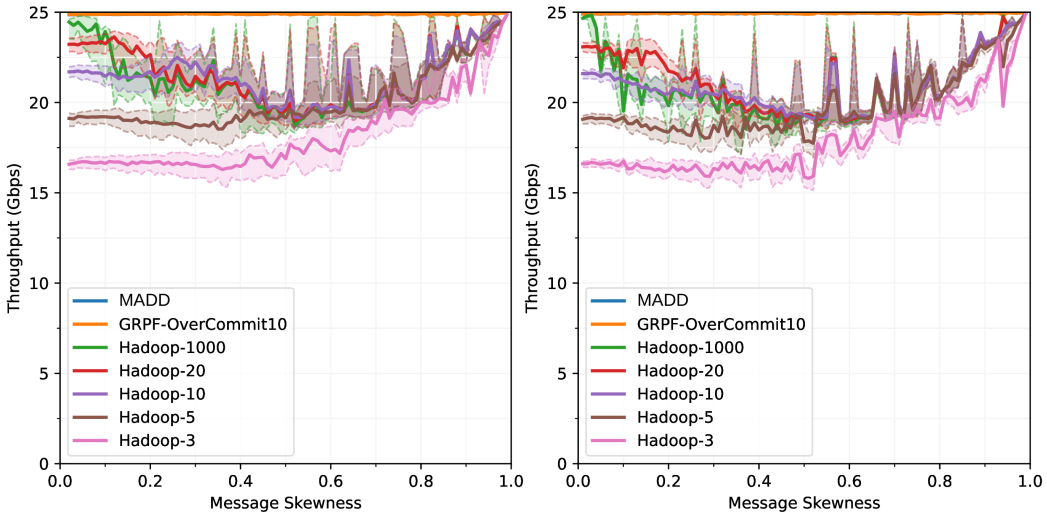
The MADD algorithm, on the other hand, is implemented verbatim by explicitly computing the optimal sending rate for each message at the beginning of shuffle. Note that MADD is not applicable to flash bursts in practice due to its more centralized nature (this is not a problem in simulation), and it needs to know the underlying network topology a priori in the case of limited core bandwidth.

The rest of the chapter is organized as follows. First, Section 6.1 evaluates the various algorithms under an ideal environment. Then, for non-ideal environments, I considered two cases: (i) application preemption due to competing workloads (Section 6.2) and (ii) limited core bandwidth (Section 6.3). The evaluation of bandwidth variation in the network is left to future work; my hypothesis is that this is less challenging than the case of limited core bandwidth, especially if the bandwidth variation follows the same pattern for all links.

6.1 Ideal environment

In an ideal environment, there is no interference from competing workloads, and the network has full bisection bandwidth. There are two major goals of this section. First, it evaluates the performance of various algorithms operating under an ideal environment. Second, it evaluates the performance benefits of individual mechanisms employed by the GRPF algorithm by starting with a naïve receiver-driven protocol and gradually transforming it into the full-fledged GRPF algorithm.

[1]The "high-entropy" approach (Section 2.4.1) used by MilliSort is close to the Hadoop algorithm with an unlimited number of concurrent connections; the major difference is that it applies randomization instead of fair sharing when scheduling messages.



(a) 4x40 nodes **(b)** 8x40 nodes

Figure 6.1: Shuffle throughput of different algorithms as a function of message skewness. The two graphs simulate a total of 4 and 8 racks, respectively (each rack has 40 nodes). The average message size is 16 full packets (the optimal completion time in the right graph is about 2.46 ms). Each curve denotes a different algorithm (`Hadoop-`C means each receiver can have at most C incoming messages concurrently). The experiment runs 100 simulations for each value of x. The solid lines indicate the median throughput, while the shaded regions indicate the 10th and 90th percentiles.

First and foremost, it's worth noting that all experiments in this chapter use throughput to measure the shuffle performance. More precisely, the throughput of a shuffle is defined as the total bytes transmitted over the shuffle divided by the number of nodes and the time to receive the last message: i.e., $\frac{total\_data\_bytes}{num\_nodes \times time\_to\_recv\_last\_msg}$ (it's neither the average throughput of individual nodes nor the throughput of the slowest node).

Figure 6.1 shows a performance overview of the various algorithms of interest. MADD can achieve 25 Gbps easily regardless of the message skewess (the blue curve is barely visible at the top of the graph); this is expected because its optimality has been formally proved. The GRPF algorithm is also extremely close to the maximum throughput (the orange curve is almost indistinguishable from the blue one), as it's able to enforce the pro rata rate allocation almost perfectly; later experiments will show how various pieces of the algorithm are combined to achieve this in a synergistic way. Finally, no configuration of the Hadoop algorithm is able to achieve satisfying performance; Hadoop-5, the configuration used for evaluation in Chowdhury et al. [14], leaves 20–30% of the throughput on the table for a wide range of message skewness up to 0.7. On a side note, the Hadoop curves have some significant spikes occasionally. This is because the shuffle workload generator sometimes produces patterns with very few non-zero messages for these message skewness, and these patterns are much easier to schedule (thus, the sudden increase in throughput).

Finally, it's worth noting that the notations here will be used repeatedly in the rest of the chapter.

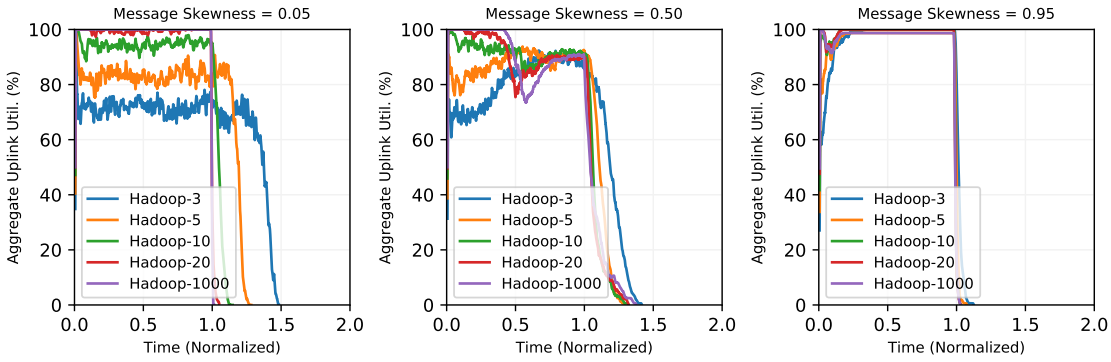


Figure 6.2: Aggregate uplink bandwidth utilization as a function of time (normalized to a value of 1.0 for the optimal completion time). From left to right, the experiments were run with message skewness 0.05, 0.50, and 0.95, respectively. All experiments simulate a total of 160 nodes.

Thus, unless said otherwise, when an experiment needs to show the shuffle throughput as a function of message skewness, it will run at least 100 simulations for each message skewness and denote the 10th, 50th, and 90th percentiles in the same way as Figure 6.1.

6.1.1 Problems of the Hadoop algorithm

The biggest problem of any fair-sharing scheme such as the Hadoop algorithm is that senders with fewer numbers of active outgoing messages often have to under-utilize their uplink bandwidth. For example, when each receiver grants to 5 random senders, some senders will be granted fewer than 5 messages due to randomness. If a sender happens to be granted only one message, then there is a high chance that this sender will have to waste 80% of its uplink bandwidth because the receiver has to allocate its downlink bandwidth evenly across its 5 incoming messages (the sender may only receive more than 20% of the downlink bandwidth if some other senders of this receiver decide not to use up their quota). As C, the number of concurrent incoming messages per receiver, increases, the probability for a sender to be granted significantly fewer messages than C decreases, and thus the average bandwidth wastage of the most unlucky sender also drops.

In order to better visualize the problem, Figure 6.2 picks three workloads with low, medium, and high message skewness, respectively, from the 160-node experiment and shows the aggregate uplink utilization as a function of time. Although the aggregate uplink bandwidth improves consistently as C increases from 3 to 20, especially for the cases of low and medium message skewness, the return of increasing C diminishes very quickly. As shown in Figure 6.1, even with $C = 1000$, the performance is still far from optimal in most cases; in fact, Hadoop-1000 only outperforms Hadoop-20 for very low message skewness and has either the same or worse throughput than Hadoop-20 in other cases.

The middle graph of Figure 6.2 provides some insights into the limit of increasing C. While the

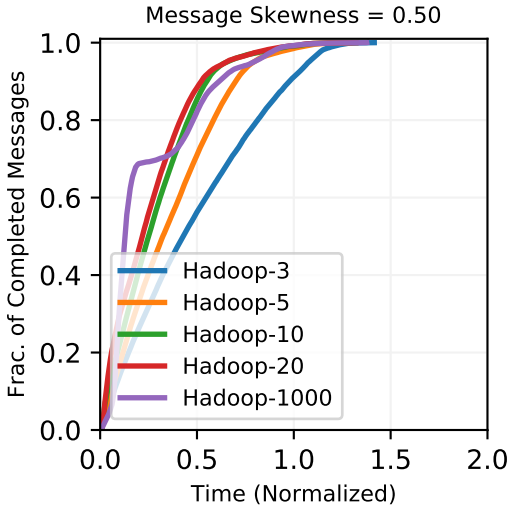
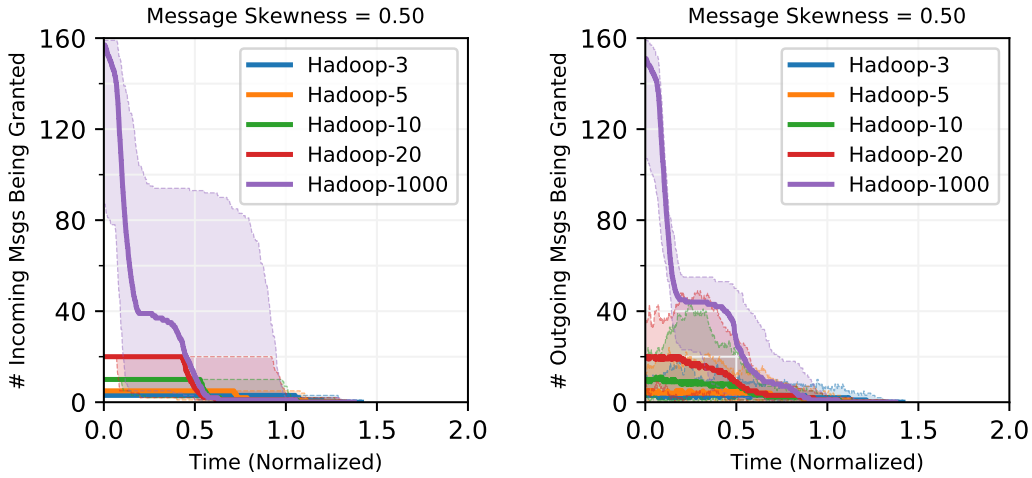


Figure 6.3: Fraction of messages completed as a function of time. The data are collected from the same experiment as the middle graph of Figure 6.2 (i.e., the message skewness is 0.50).

aggregate uplink utilization of Hadoop-1000 starts up as the highest among all configurations, it quickly drops to about 72%, which is even worse than Hadoop-3, before it gradually goes up again; the net result is that the completion time of Hadoop-1000 is still roughly the same as Hadoop-10.

Further investigation reveals that the rapid drop in aggregate uplink utilization is caused by a combination of two factors. First, when C is larger than the size of the cluster, every receiver grants to all senders simultaneously, and the fair-sharing policy will complete small messages early in the shuffle. Figure 6.3 shows that Hadoop-1000 completes roughly 70% of the messages in less than $\frac{1}{5}$ of the total shuffle time (this also implies that 70% of the messages are rather small in this workload). Thus, the actual numbers of incoming and outgoing messages currently being granted at a node will soon be limited by the number of unfinished messages. Second, it turns out that the shuffle patterns produced by the workload generator has the following characteristic at medium message skewness: once the small messages complete, the numbers of remaining messages at receivers vary significantly across nodes, while the numbers of remaining messages at senders are much more uniform. Figure 6.4 clearly shows this pattern: e.g., consider the Hadoop curves at time equals 0.25, the number of remaining messages at receivers ranges from 5 to 95, while the number of remaining messages at senders is between 20 and 55. This pattern is especially challenging to the fair-sharing policy: receivers that have fewer messages left often end up under-utilizing their downlink bandwidth, since their senders often have to allocate uplink bandwidth across much larger numbers of remaining outgoing messages; conversely, receivers that have larger numbers of messages left often cannot grant fast enough to keep all of their senders busy simultaneously. Thus, while the network utilization of



(a) Incoming messages at receivers.

(b) Outgoing messages at senders.

Figure 6.4: Numbers of incoming (outgoing) messages currently being granted at different nodes as a function of time (the solid lines denote the median; the shaded regions denote the min and max values). The data are obtained from the same experiment as the middle graph of Figure 6.2 (i.e., the message skewness is 0.50).

Hadoop-1000 in the middle graph of Figure 6.2 achieves 100% in the beginning, thanks to the initial RTTpackets grants per message, it drops significantly once more under-granted senders start to run out of grants. Note that over time, as more messages complete (i.e., only large messages are left), the shuffle pattern becomes less pathological for the fair-sharing policy, and the network utilization starts to rise again; intuitively, the impact of C decreases with the number of remaining messages, so different curves in the middle graph of Figure 6.2 gradually converge towards the end of shuffle.

6.1.2 Scheduling policy

In order to evaluate the various features of the GRPF algorithm independently, a simplistic receiver-driven protocol will be used as the baseline, and then each of the following sections will add back one mechanism at a time.

The baseline protocol is extremely simple and agnostic to scheduling policies. In each timestep, each sender scans its outgoing messages that have grants available and selects the next message to transmit based on some policy (e.g., GRPF). Further, senders are not allowed to transmit any data without first receiving grants from receivers (i.e., no unscheduled packets). The receivers apply the same policy in deciding which messages to grant next. Each receiver can send out more than one grant in each timestep, and it always attempts to maintain a total of one RTT worth of outstanding grants (i.e., no overcommitment) across all its incoming messages. Each message also has a constant

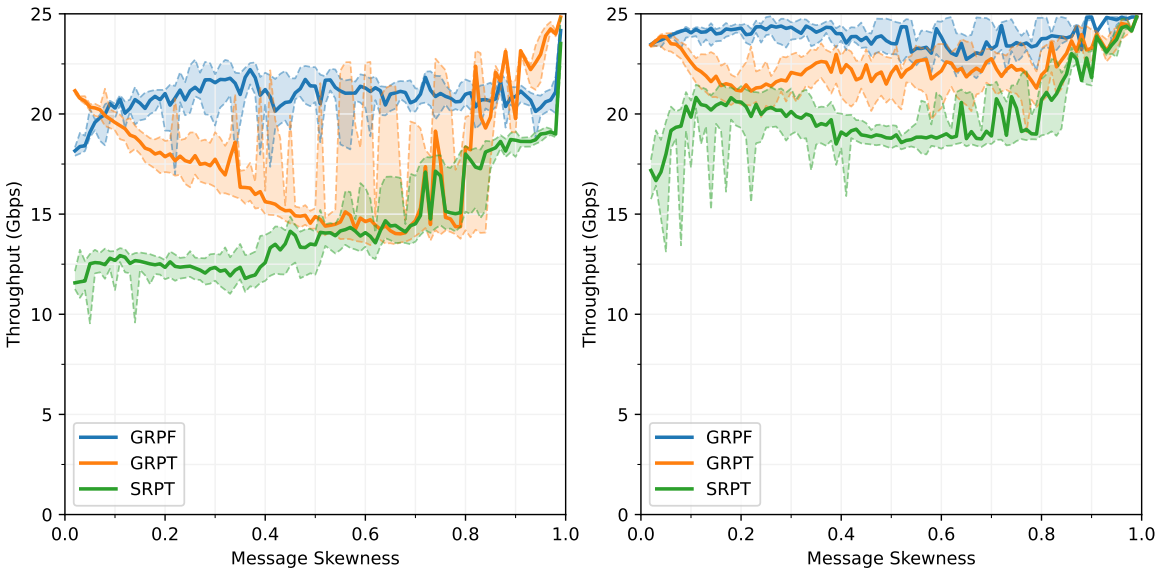
grant window limit of RTTpackets; however, this limit has no effect in the baseline protocol because the maximum total outstanding grants of a receiver is also RTTpackets. Thus, the receiver must always grant to the top RTT worth of incoming packets (as determined by the scheduling policy); unlike the full-fledged algorithm, it cannot direct its grants elsewhere even if its top-priority senders decide to transmit to someone else. Finally, grants are transmitted with the same network priority level as data packets.

The rest of this section evaluates the choice of the scheduling policy; Section 6.1.3 overcommits the receiver downlinks to reduce the likelihood of running out of grants at senders; Section 6.1.4 adds per-message pro rata grant window limit to solve the performance regression due to overcommitment (and discusses why network priority is not necessary in an ideal environment); finally, Section 6.1.5 analyzes how randomization in the scheduling policy can be used to mitigate performance drops due to micro-incasts.

Two more scheduling policies are of interest in addition to GRPF. First, SRPT (shortest remaining processing time first) is a well-known policy that provides near-optimal average message latency. An SRPT sender prioritizes packets from messages with the fewest bytes remaining to transmit, and an SRPT receiver prioritizes grants to messages with the fewest bytes remaining to receive. Second, GRPT (greatest remaining processing time first) is the exact opposite of SRPT, which always favors messages with the most bytes remaining to transmit or receive. In contrast to GRPF, GRPT is only concerned with the absolute remaining sizes of messages, as opposed to the relative remaining sizes (i.e., the remaining fractions). In fact, GRPT was the policy of choice in the initial design because it handles straggler messages much better than SRPT.

Figure 6.5a shows the performance of the baseline protocol with different choices of the scheduling policies. It is not surprising that SRPT has the worst performance out of the three policies, since it optimizes for the completion times of individual messages, as opposed to the completion time of the last message in the shuffle. More specifically, SRPT often fails to achieve high network utilization for two reasons. First, SRPT may lead to mismatched priorities between senders and receivers: the shortest incoming message of a receiver is not necessary the shortest outgoing message of its sender. When a mismatch occurs, the receiver's downlink bandwidth is wasted. Correspondingly, this is also problematic on the sender side: if all the outgoing messages of a sender are large, then it may not receive grants from the receivers until much later. Second, to make things worse, since the baseline protocol doesn't limit the number of grants that can be issued to a message, SRPT receivers will enforce a grant-to-completion behavior on the shortest messages, which produces a more persistent mismatch between senders and receivers.

GRPT outperforms SRPT in all cases but it still leaves significant performance on the table. Unlike SRPT, GRPT doesn't enforce a run-to-completion behavior; quite the opposite, GRPT will eventually degenerate to round robin when the remaining parts of all messages are equally long. In practice, this means a GRPT receiver usually only grants to a small subset of the incoming messages



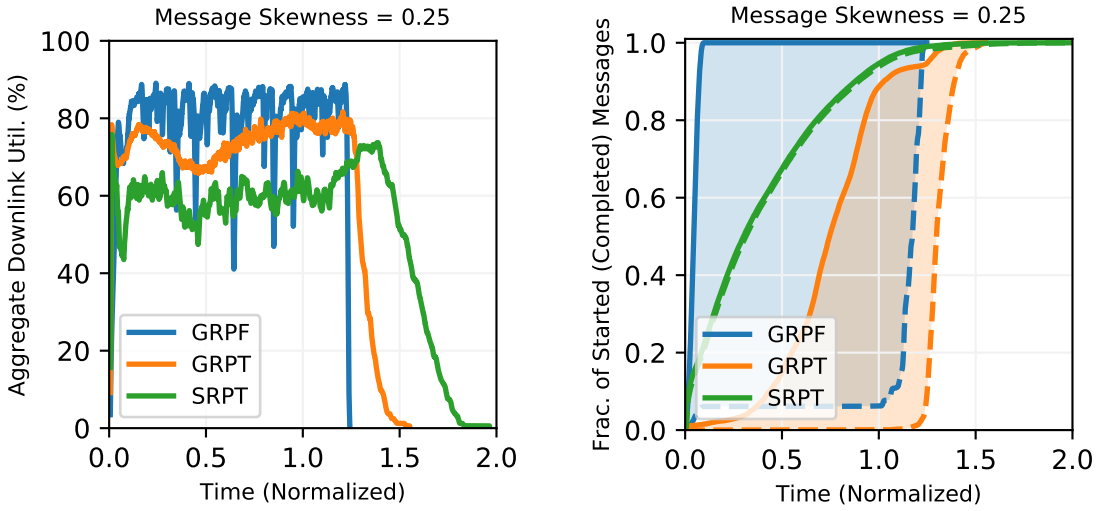
(a) Baseline protocols **(b)** Overcommitment + network priorities

Figure 6.5: **(a)** shows the shuffle throughput of three different scheduling policies as a function of time. **(b)** adds an overcommitment level of 10 to all three baseline protocols and assigns grants a higher network priority than data packets. All experiments simulate a total of 160 nodes.

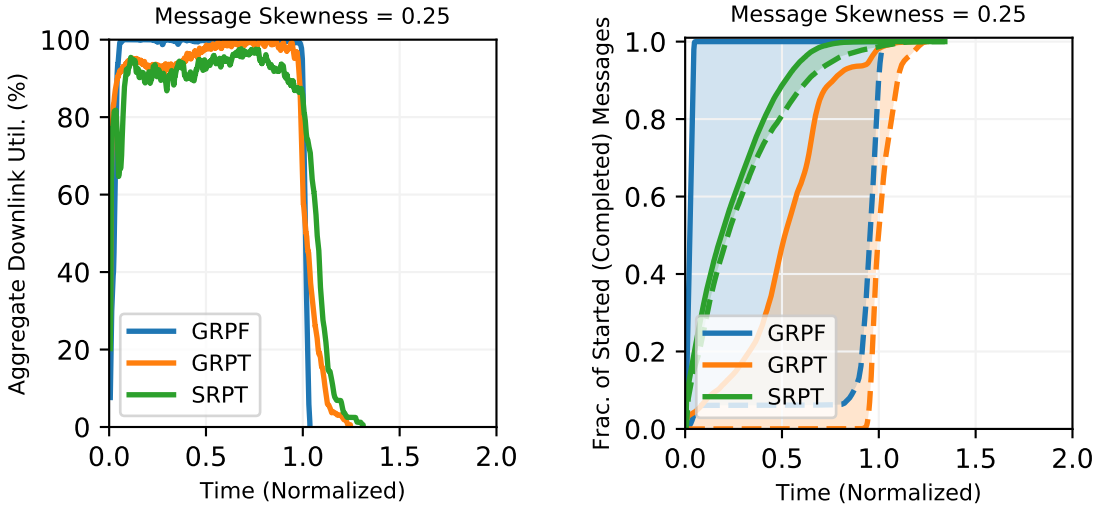
in the early stage of shuffle, and then it gradually includes more incoming messages in its active set when more messages become equally long. While this behavior is more desirable than SRPT, there is no guarantee that GRPT will be able to fully utilize the network eventually. For example, the aggregate downlink utilization of GRPT in the left graph of Figure 6.6a never surpasses 85%. This is because, with a message skewness of 0.25, nodes will have very different numbers of non-empty messages; as discussed in Section 6.1.1, the round robin (fair-sharing) policy can waste significant bandwidth in this case. Finally, GRPT does not solve the problem of mismatched priorities between senders and receivers: the longest incoming message of a receiver is not necessary the longest outgoing message of its sender. As a result, GRPT is strictly inferior to GRPF.

Although GRPF has the best performance out of the three policies, the resulting baseline protocol is still far from optimal. There are two main observations from the left graph of Figure 6.6a: (i) the maximum network utilization is still capped at around 90%, and (ii) the network utilization suffers from periodic sudden drops over the course of shuffle. They will be discussed below in Section 6.1.3 and 6.1.5, respectively.

To better visualize the behaviors of the different policies, Figure 6.6a further measures the fractions of started and completed messages as a function of time for all three policies using the same workload. The two green curves of SRPT in the middle and right graphs have very similar shapes (both rise rapidly early in the shuffle), which confirms the run-to-completion behavior of SRPT.



(a) Baseline protocols



(b) Overcommitment + network priorities

Figure 6.6: Behaviors of different protocols for a specific workload with a message skewness of 0.25. The two rows correspond to the two different protocol configurations in Figure 6.5, respectively. The left graph of each row shows the aggregate downlink utilization as a function of (normalized) time. The right graph of each row shows the fractions of messages started (in solid lines) and completed (in dashed lines) as a function of (normalized) time.

The orange curve of GRPT in the middle shows that GRPT senders are working on a small set of outgoing messages early in the shuffle (less than 20% of the messages have been selected for transmission at $t = 0.50$). GRPT also has much fewer straggler messages than SRPT (almost all messages complete at around $t = 1.25$), but it wastes too much bandwidth compared to GRPF. Finally, the blue curves of GRPF exhibit a rather desirable behavior: all messages are eligible for transmission from the very start, yet, unlike SRPT, the majority of them will remain unfinished until the end of shuffle (a small caveat here is that GRPF finishes single-packet messages much earlier than GRPT because they have the same scheduling priority as other messages at the beginning of shuffle; this is a relatively small issue, and it will be addressed by randomization in Section 6.1.5 below).

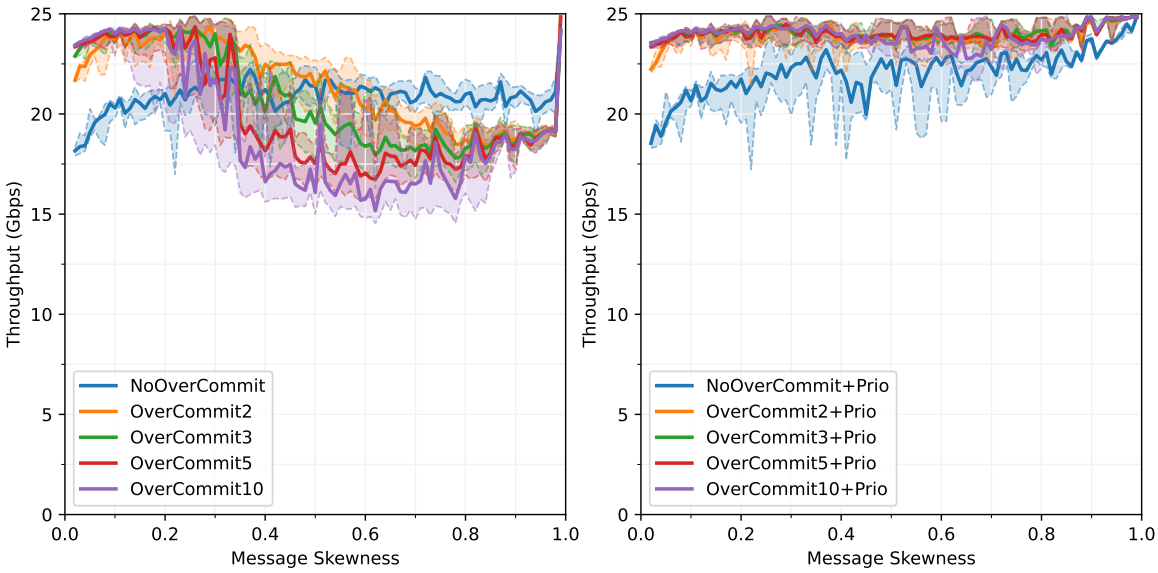
One drawback of the analysis above is that it ignores the impacts other design features might have on the scheduling policy. It is possible that the lack of certain features in the baseline protocols creates more problems for GRPT and SRPT than GRPF. For example, Homa [69] has shown that it can easily sustain 80–90% network load for several common workloads in datacenters despite using SRPT. Thus, in order to conduct a fair comparison, Figure 6.5b reruns the experiments after adding overcommitment to all three policies and assigning grants a higher priority than data packets (this is probably not the best possible configuration for GRPT and SRPT, but a reasonable one nonetheless). Further, Figure 6.6b shows the behaviors of different policies for a specific workload with a message skewness of 0.25 with the new configuration. While the use of overcommitment and network priorities improves the aggregate network utilization of GRPT and SRPT effectively, the conclusion from the previous analysis remains true; in particular, the shapes of the curves in the middle and rightmost graphs of Figure 6.6b are almost the same as those in Figure 6.6a.

6.1.3 Overcommit

Further investigation reveals that the throughput of GRPF in Figure 6.5a is limited by the number of outstanding grants. After all, one RTT worth of outstanding grants is barely enough to cover the communication latency between senders and receivers in the ideal case. As a result, a sender has to waste its uplink bandwidth whenever a grant is delayed, or when its receivers decide to favor other senders simultaneously.

Thus, the next step is to enable overcommitment in the protocol. Figure 6.7a shows the performance of various degrees of overcommitment, and the results turn out to be mixed. At low message skewness, overcommitment improves the shuffle throughput significantly. However, as the message skewness increases, the benefit of overcommitment drops and even becomes negative eventually; at which point, the more the receivers overcommit, the worse the performance becomes.

The performance regression resulting from overcommitment is an interesting secondary effect of the increased buffer buildups at the ToR switches. Overcommitment essentially grants senders more freedom in allocating their uplink bandwidth. Thus, if a sender decides to transmit a message at a higher rate than the receiver desires (i.e., there is a discrepancy between the scheduling decisions



(a) Without network priorities. **(b)** With network priorities.

Figure 6.7: Performance impact of overcommitment. `NoOverCommit` is the same as `GRPF` in Figure 6.5a. `OverCommitN` allows each receiver to have a maximum of N RTT worth of outstanding grants. `OverCommitN+Prio` is similar to `OverCommitN` except that it assigns grants a higher network priority than data packets.

made on two sides of the message), the total incoming data rate can exceed the downlink capacity of the receiver temporarily and build up a queue at the switch. Now all packets (including grants) destined to the receiver will suffer a queuing delay; if the grants are delayed for too long, then the colocated sender of the congested receiver will be starved of grants and waste uplink bandwidth. As a side note, the regression is more severe at high message skewness. This is because it is more likely to have nodes that send much less data than others at higher message skewness (recall that uniform data distribution doesn't entail the exact same amount of outgoing data for every node due to the existence of local data; Section 5.1); with GRPF, these senders usually end up sending data faster than the receivers desire and cause more buffer buildups.

This problem is responsible for most of the performance regression. A simple strawman solution is to eliminate the queuing delay of grants completely by assigning them a higher network priority than data packets, and it's quite effective as displayed in Figure 6.7b; this figure also shows that an overcommitment level of two is sufficient in an ideal environment. However, the use of the in-network priority may limit the applicability of the algorithm in certain environments. Fortunately, there is a better solution, pro rata grant window limit, which is essential to the full-fledged algorithm anyway.

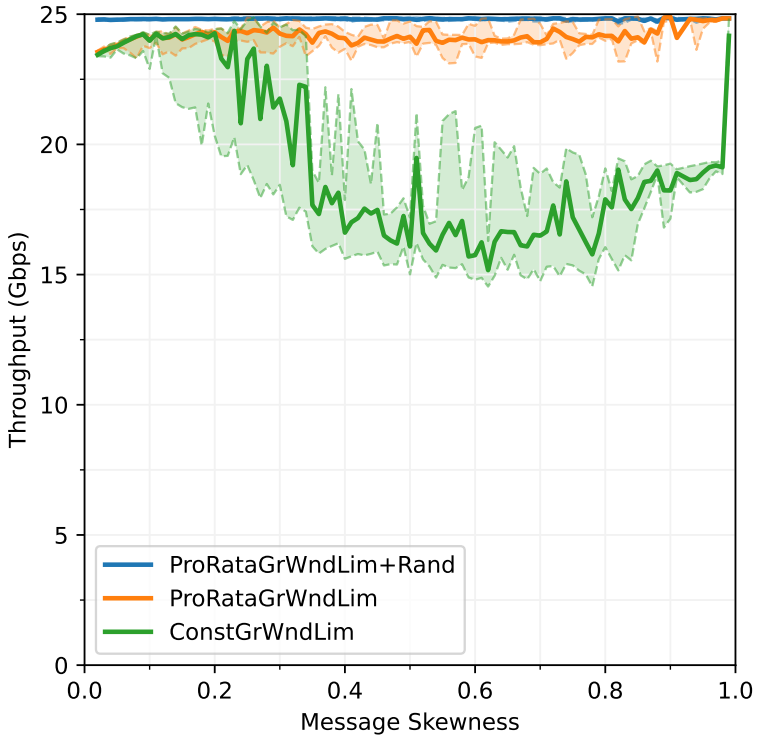


Figure 6.8: Performance gains of pro rata grant window limit and randomization. `ConstGrWndLim` is the same as `OverCommit10` in Figure 6.7a. `ProRataGrWndLim` applies pro rata grant window limit on top of `OverCommit10`, and then `ProRataGrWndLim+Rand` further adds randomization in the GRPF policy. No network priority is used in any of these experiments. The results of other overcommitment levels are similar, so they are omitted here (similar to Figure 6.7b, overcommitment beyond 2x makes almost no difference for shuffles running in an ideal environment).

6.1.4 Pro rata grant window limit

So far each message has a fixed grant window limit of RTTpackets that is independent from the total outstanding grants of a receiver. As discussed in Section 3.1.6, a constant grant window limit cannot uphold the pro rata rate allocation scheme when one or more messages are being transmitted faster than they are expected by the receiver; these messages will end up consuming more donwlink bandwidth than they should. Pro rata grant window limit solves this problem by scaling the per-message grant window limit with both the message length and the degree of overcommitment.

A pleasant side effect of pro rata grant window limit is that it solves the performance regression resulting from overcommitment in the previous section elegantly without using in-network priority. Figure 6.8 shows that it achieves the same level of performance as `OverCommit10+Prio` in Figure 6.7b. This is possible because the buffer buildups at the ToR switches and the total incoming grants of individual senders now increase together with the degree of overcommitment. For example, when a receiver increases its maximum outstanding grants by RTTpackets (i.e., overcommits the downlink

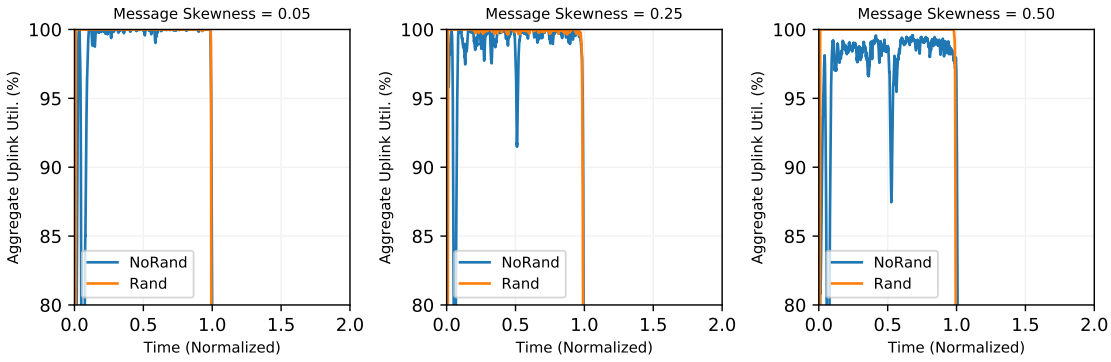


Figure 6.9: Aggregate uplink bandwidth utilization of three specific workloads before and after applying randomization. From left to right, the workloads have message skewness 0.05, 0.25, and 0.50, respectively. `NoRand` is the same as `ProRataGrWndLim` in Figure 6.8. Note that the y-axes on these graphs are not zero-based.

by 2x), the worst-case queue length in front of its downlink also increases by RTTpackets (i.e., any incoming grant can be delayed by at most RTT at the ToR). In the meantime, consider the sender that colocates with this receiver, its total incoming grants also increases by RTTpackets, so it can tolerate RTTpackets grants being queued in the network without running out of grants. As a result, the senders are able to fully utilize their uplinks regardless of the degree of overcommitment.

6.1.5 Randomization

To understand why the current protocol is still not achieving 25 Gbps, Figure 6.9 picks three specific workloads and plots the aggregate uplink utilization as a function of time. Similar to the leftmost graph of Figure 6.6a, periodic drops in network utilization occurred over the course of shuffle.

Detailed analysis of packet traces reveals that the periodic drops can be attributed to the naive deterministic implementation of GRPF. As discussed in Chapter 3, when two incoming messages are of the same size, corresponding packets from the messages will also be assigned the same scheduling priority. Further, when two incoming packets have the same scheduling priority, they will likely be sent to the receiver at roughly the same time. Thus, if many messages are of the same sizes, then periodic incasts will be inevitable; these incasts are very bursty in nature, so I refer to them as micro-incasts. As a result of the incast, some other receivers will waste downlink bandwidth, and the senders of the incast will not be able to receive grants in time later (the target receiver of the incast cannot receive the packets fast enough to send back grants).

Randomization solves this problem by picking a random offset within each packet and using that offset to compute the corresponding scheduling priority (i.e., $sched\_priority = 1 - \frac{rand\_offset}{message\_len}$). This simple trick eliminates almost all micro-incasts. Figure 6.8 shows that the resulting throughput is very close to the line rate of 25 Gbps.

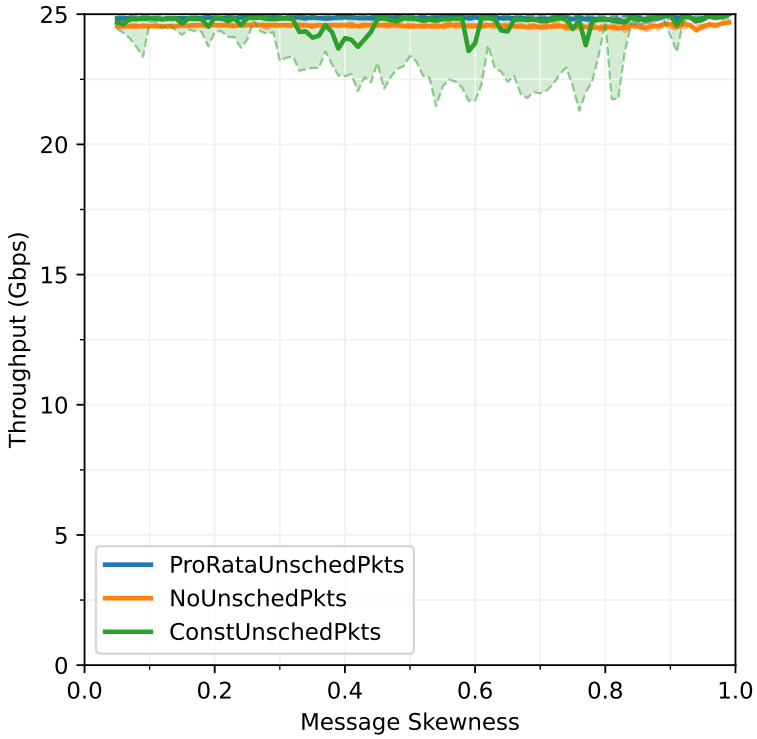


Figure 6.10: Performance impact of unscheduled packets. `ProRataUnschedPkts` enforces one RTT worth of unscheduled packets for each sender and allocates them among each sender's outgoing messages pro rata. `ConstUnschedPkts` allows the first RTTbytes of each message to be unscheduled. Note that this experiment simulates a smaller cluster of 40 nodes; the average message size is still 16 packets.

6.1.6 Pro rata unscheduled packets

The protocol so far is already near-optimal for relatively long-running shuffles. However, for lower-latency shuffles that have less data per node, it would be nice for each sender to be able to transmit some data unilaterally. There are two ways to configure the amount of unscheduled packets. One is to make the first RTTbytes of each message unscheduled like Homa [69]; the other is for each sender to enforce a total of one RTT worth of unscheduled packets, and allocate them across its outgoing messages in proportion to the message lengths.

Figure 6.10 shows the performance of the two approaches. In order to better display the difference, this experiment reduces the cluster size to 40 nodes. The pro rata approach is able to improve the performance consistently without any regression; its performance benefit will be more significant if the amount of data per node is even smaller. On the other hand, one RTTbytes unscheduled traffic per message is way too aggressive for the receivers to enforce their pro rata rate allocation scheme accurately; this results in up to 15% performance loss at the 90th percentile.

6.1.7 Network priority

As discussed in Section 6.1.4, the GRPF algorithm can achieve near-optimal performance in an ideal environment without in-network priority. However, when buffer buildups become more significant in the network (e.g., Section 6.3), allowing grants to bypass packet queues in the switches will become essential to achieving optimal performance.

In practice, enabling the support for in-network priority at the edge of the network (i.e., output ports directly connected to the end hosts) is easier than at the core when the end hosts are dedicated to a single user, since modifying the configuration of these ports will only affect traffic pertaining to that user. This configuration is enough until Section 6.3, where I will show why in-network priority must also be enabled in the network core for best performance.

6.2 Application preemption

In order to improve hardware utilization, datacenter operators often multiplex CPU cores between different applications. Thus, I expect the shuffle algorithm to be perturbed by application preemption frequently. In the simulation, when the shuffle application running on a node is preempted, it has to stop sending or receiving any packets until it's scheduled again. Note that this won't stop the kernel networking stack or the NIC, so incoming packets will continue to be received in the host memory in the meantime.

As a result, application preemption doesn't affect the downlink utilization of a receiver directly. Further, recall that the simulator assumes the application can process incoming packets much faster than the line rate of the NIC, so a receiver will be able to catch up on the incoming grants quickly after being rescheduled and then saturate its uplink. Thus, the only impacts of preemption are (i) senders have to stop sending data packets, and (ii) receivers have to stop issuing grants; both of which cause under-utilization of the senders' uplinks.

The first issue might be addressed by having each sender queue some extra outgoing packets in the NIC to keep the uplink busy, but the current simulator has not implemented this solution yet; this is left for future work. Thus, the only issue in dealing with preemption here is how to keep the non-preempted senders as busy as possible when some receivers fail to respond.

6.2.1 Baseline performance

The observation above also suggests that one can obtain a near-optimal baseline performance via the MADD algorithm (or the GRPF algorithm with an infinitely large overcommitment), since MADD senders don't need grants at all (they simply transmit data from messages at preset rates as long as they are scheduled). Figure 6.11 shows the shuffle throughput of MADD under varying average lengths of the preemption intervals. When the average preemption interval is small, the

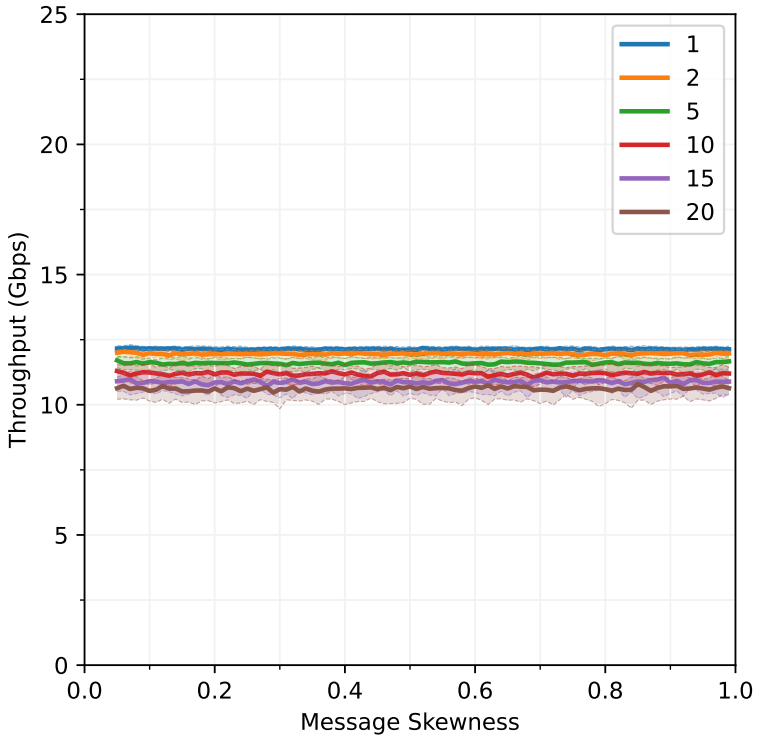


Figure 6.11: Baseline shuffle throughput (from running MADD) as a function of message skewness. The experiment simulates a total of 40 nodes under one ToR switch. Each curve denotes a different average length of the preemption intervals, measured in multiples of the round-trip time: e.g., the largest average preemption interval is 20 RTTs (the RTT within a rack is 8 timesteps). On average, each node will be preempted 50% of the time during simulation; the length of the preemption interval follows a normal distribution whose standard deviation is $\frac{1}{4}$ of the mean value. To reduce noise in measurements, the average message size is set to 400 full packets, so the optimal shuffle time without preemption is roughly 100x of the maximum average preemption interval.

shuffle throughput is very close to 12.5 Gbps, since each node is preempted roughly 50% of the time. However, as the average preemption interval becomes larger, the throughput decreases slightly. This is because the preemption schedule is generated randomly, and some nodes happen to be preempted slightly more than others. The gap between the curves will narrow if the simulation runs longer (i.e., with a larger average message size).

In order to confirm that the baseline numbers are indeed near-optimal, I also derive a theoretical upper bound on performance with three oracles. Each oracle takes as input the preemption schedule of each node during the simulation and computes some lower bounds on the shuffle completion time from the perspectives of individual senders, receivers, or messages; the greatest lower bound is then used to compute an upper bound on the shuffle throughput. The first oracle computes the minimum time for each node to transmit all its data assuming unlimited grants. The second oracle computes the minimum time for each node to receive all its data assuming no other receiver exists in the

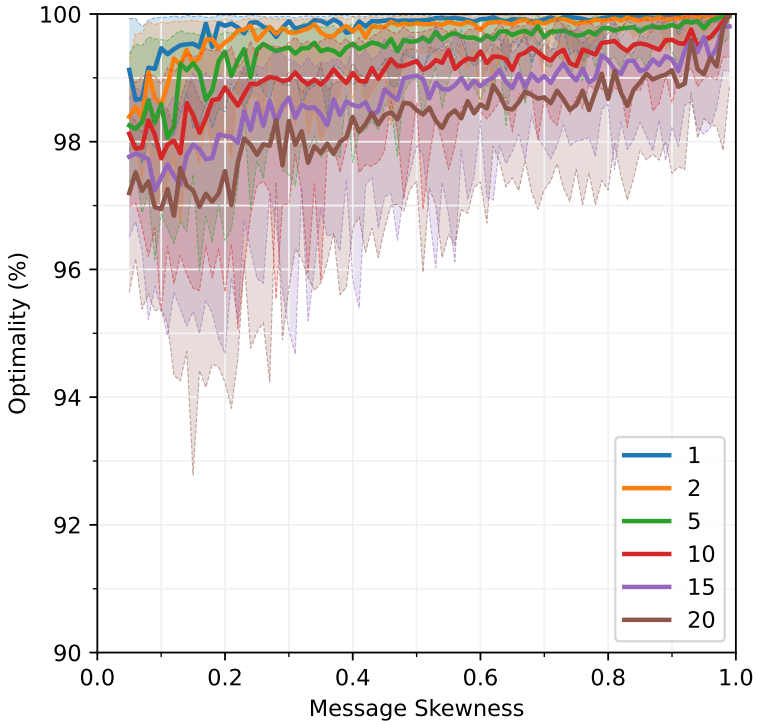


Figure 6.12: Optimality of the baseline throughput in Figure 6.11 as a function of message skewness. Optimality is defined as the ratio between the baseline throughput and the theoretical upper-bound derived from the oracles. Each curve denotes a different average length of the preemption intervals, measured in multiples of the round-trip time. Note that the y-axis is not zero-based.

system. The third oracle computes the minimum time for each message to finish assuming no other message exists in the system. For simplicity, the oracles omit the network latency in all computation.

Figure 6.12 shows the ratio between the baseline throughput and the theoretical upper bound. The baseline throughput is within 1.5% of the upper bound at the median, and no more than 5% in most cases. In addition, my hypothesis is that the baseline performance is actually even closer to the optimal than it appears, since the current throughput upper bound is not tight: there is no guarantee that the individual lower bounds on the shuffle completion time can be achieved simultaneously, so the maximum throughput is likely over-estimated. Designing a perfect oracle seems non-trivial, so the baseline throughput from running MADD will be used for comparison in the rest of the section.

6.2.2 Overcommitment and bandwidth redirection

Unfortunately, the performance achieved by MADD (or GRPF with infinite overcommitment) is not very meaningful, since transmitting data blindly from the senders can easily lead to buffer overflows in general. In practice, when a receiver becomes unresponsive, the sender has no way to tell whether this is because the network is congested or the receiver has been preempted. In the former situation,

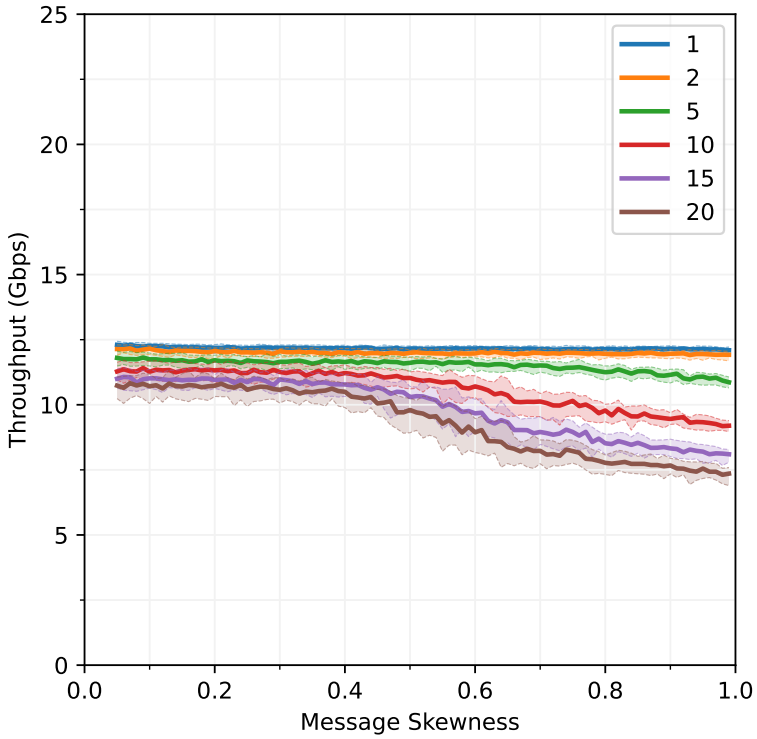
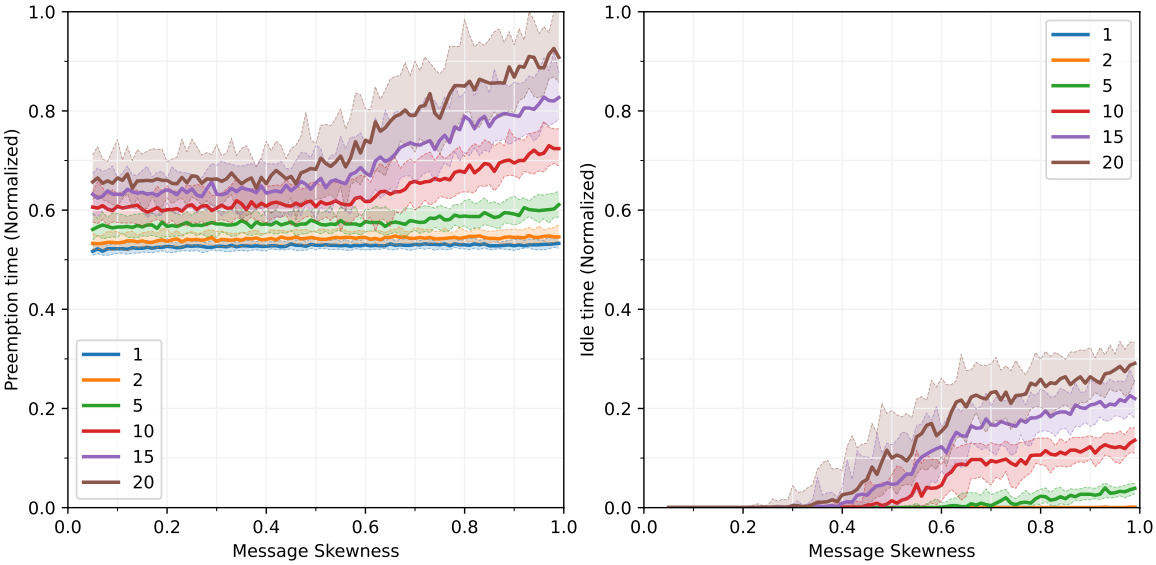


Figure 6.13: Shuffle throughput of the GRPF algorithm, with an overcommitment level of 10, as a function of message skewness. Same as Figure 6.11, each curve denotes a different average length of the preemption intervals, measured in multiples of the round-trip time.

it's a bad idea to continue sending to this receiver, since it will exacerbate the queue buildup at the congestion point. As a result, a practical implementation of the MADD algorithm must employ other mechanisms to avoid buffer overflow, which makes it impossible to stick to its precomputed sending rates.

Figure 6.13 shows the shuffle throughput of the GRPF algorithm with an overcommitment level of 10. At low to medium message skewness, there is no performance loss compared to the baselines in Figure 6.11. However, as the message skewness increases, the performance starts to drop; in the worst case, the performance loss can be more than 25% (from 10.5 Gbps to 7.5 Gbps at the highest message skewess and with an average preemption interval of 20 RTTs). The GRPF algorithm can keep up with the baseline throughput at lower message skewness because the protocol allows a sender to redirect its bandwidth elsewhere to keep its uplink busy when a receiver is not responding (and after all its overcommitted grants are exhausted). Unfortunately, as the message skewness increases, the number of non-empty messages quickly drops, so the benefit of bandwidth redirection diminishes: it becomes increasingly hard for a sender to find an active receiver to absorb its unused bandwidth, especially at a later stage of the shuffle when more messages are completed.



(a) Time for which the straggler is preempted. **(b)** Time for which the straggler has no grant avail.

Figure 6.14: Two factors responsible for uplink under-utilization of the slowest sender (i.e., the straggler). Each curve denotes a different average length of the preemption intervals, in multiples of the RTT. The y values of both graphs are normalized so that a value of 0.5 is the minimum time to finish the shuffle without preemption (and a value of 1.0 if nodes are preempted 50% of the time).

Figure 6.14 measures the bandwidth wastage of the slowest sender in the shuffle that is caused by preemption and the lack of grants, respectively. When the average preemption interval is smaller than 5 RTTs, the normalized preemption time of the slowest sender is only slightly above 0.5, and the slowest sender almost never runs out of grants. However, for larger average preemption intervals, the slowest sender starts to spend more time waiting for grants as the the message skewness increases (fewer receivers exist to direct the bandwidth to); in addition, since the slowest sender takes longer time to complete, it also experiences more preemption.

Figure 6.15 further investigates the benefit of bandwidth redirection by fixing an average preemption interval of 20 RTTs and limiting the degree of bandwidth redirection as follows. Recall that, on the sender-side, each GRPF sender computes a packet schedule that contains all outgoing packets in descending order of their scheduling priorities. Usually the sender will pick the first unsent packet in the schedule to transmit; however, when that packet is destined to a receiver that is not responding, the sender may have to search deeper in the list to find the first packet that has been granted. The distance between the chosen packet and the *first unsent packet* provides a way to define the degree of bandwidth redirection. By limiting how far the sender can search down the schedule for the next packet, Figure 6.15 illustrates that a large degree of bandwidth redirection is indeed essential to achieving performance on a par with the baselines at low to medium message skewness.

In general, the larger the degree of overcommitment, the longer preemption intervals senders can

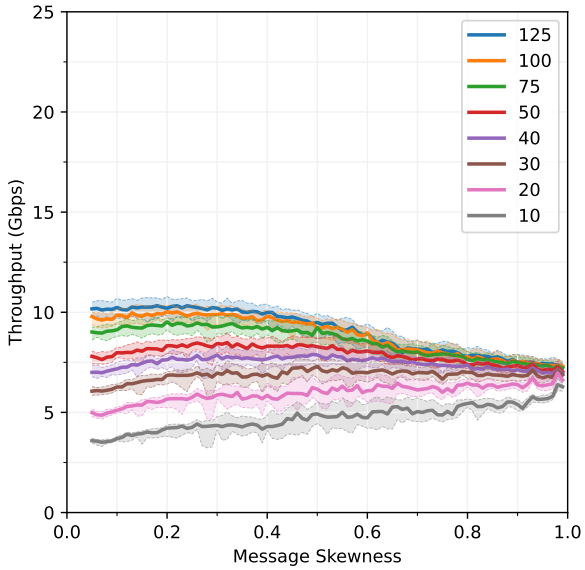


Figure 6.15: Performance gains of sender-side bandwidth redirection. The label of each curve shows the maximum distance permitted, in multiples of RTTpackets, between the first unsent packet and the next packet to send: e.g., the blue curve sets a maximum distance of $125 \times 8 = 1000$. The average length of the preemption interval is set to 20 RTTs, same as the bottom curve in Figure 6.13.

withstand without wasting uplink bandwidth. This is because a larger degree of overcommitment provides more grants for the sender to burn through; if the preempted receiver can be rescheduled in time to replenish the grants at the sender, the sender will never have to redirect (or, worse, waste) its bandwidth. Figure 6.16 fixes the average length of the preemption interval and varies the degree of overcommitment instead. This experiment shows that increasing the degree of overcommitment improves the shuffle throughput at medium to high message skewness, although the return is diminishing. When the degree of overcommitment becomes twice as large as the average preemption interval, the GRPF algorithm reaches roughly the same level of performance as MADD. In practice, such a large degree of overcommitment is probably infeasible due to the high risk of buffer overflow; this is especially true when there is competing traffic in the background or the network has limited core bandwidth. Fortunately, highly skewed workloads are likely less common in practice.

The performance of the Hadoop algorithm is less interesting. Figure 6.17 shows the throughput with various average lengths of the preemption intervals. Compared to Figure 6.13, the performance of Hadoop-10 is always worse than GRPF with an overcommitment level of 10 (the trends of the curves are similar though). In the worst case, the throughput of Hadoop-10 is lower than GRPF by almost $\frac{1}{3}$ (5 Gbps vs. 7.5 Gbps) at the highest message skewness when the average preemption interval is 20 RTTs. This is because in this workload each sender usually has only one large message that is responsible for most of its outgoing data; while GRPF-OverCommit10 may issue almost 10 RTTpackets outstanding grants to each of these large messages, Hadoop-10 only gives each message

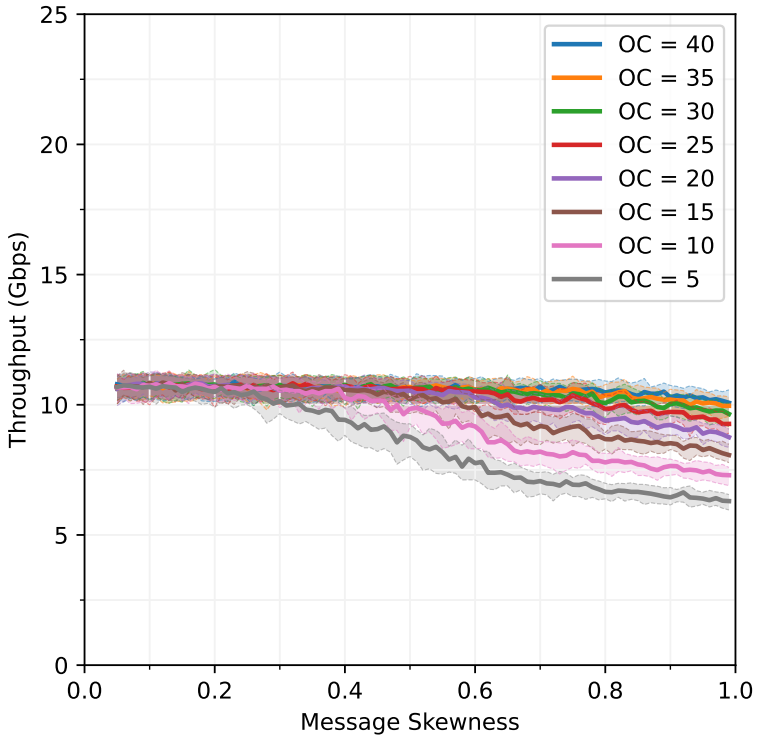


Figure 6.16: Performance gains of larger degrees of overcommitment. The average length of the preemption interval is fixed at 20 RTTs. Each curve denotes a different degree of overcommitment. All other setups follow the experiment in Figure 6.13.

RTTpackets outgoing grants, so the senders in Hadoop-10 end up under-utilizing uplinks more when their receivers are preempted.

6.2.3 Buffer occupancy

Finally, sender-side bandwidth redirection has an interesting effect on the buffer occupancy at the ToR switches: it can cause large-scale incasts towards receivers that have just recovered from preemption. This is because the GRPF policy requires senders to prioritize their bandwidth to messages that fall behind; once the grants sent by the rescheduled receiver arrive at its senders, they will simultaneously direct their bandwidth towards this receiver. The queue buildup is eventually limited by the degree of overcommitment of the receiver: in the worst case, all the overcommitted grants of a receiver will become inbound packets queued at the ToR. Careful examination of the time traces confirms that almost all (non-trivial) queue buildups during the simulation are due to bandwidth redirection.

Figure 6.18a shows that the worst-case total buffer occupancy during the shuffles increases with the average preemption interval: the longer the senders have to redirect their bandwidth, the more

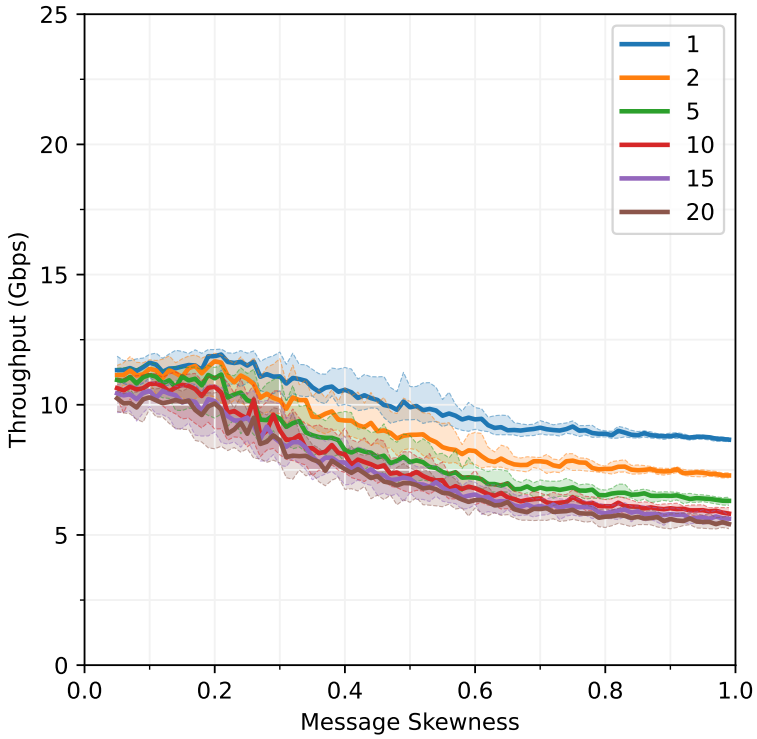
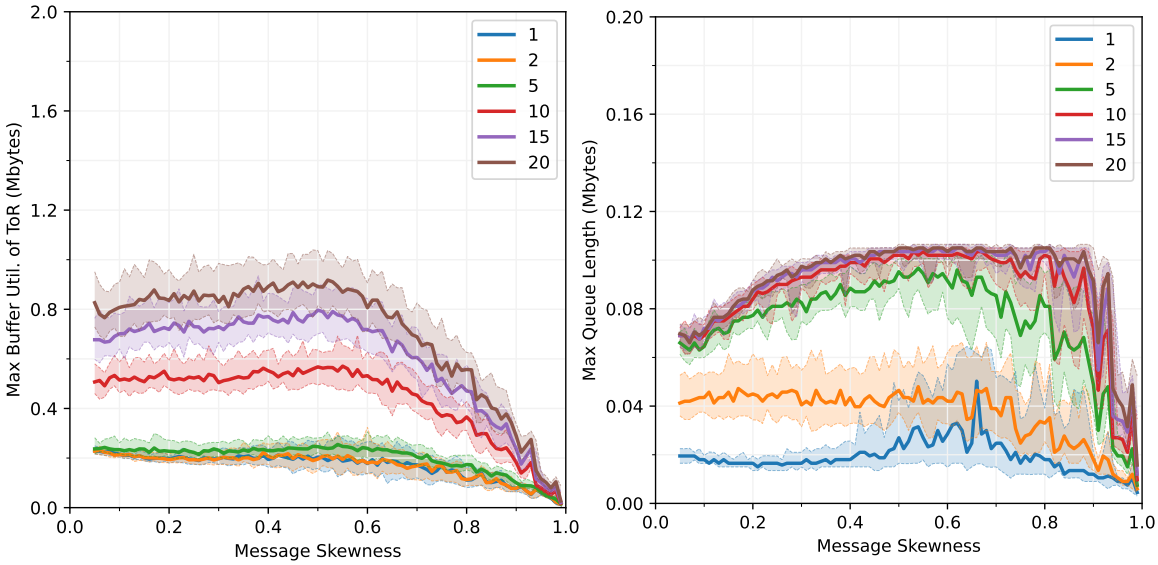


Figure 6.17: Shuffle throughput of the Hadoop-10 algorithm as a function of message skewness. Same as Figure 6.11, each curve denotes a different average length of the preemption intervals, measured in multiples of the round-trip time.

inbound data the preempted receiver needs to catch up after it recovers. As the message skewness increases, the worst-case buffer occupancy remains roughly flat at first, and then starts to decrease after the message skewness exceeds 0.5; this is because the opportunity for bandwidth redirection diminishes as the message skewness becomes too high.

The worst-case total buffer occupancy also increases with the average length of the preemption intervals. When the preemption intervals are much smaller than the maximum outstanding grants per receiver, the senders will usually have sufficient grants that allow them to stick to their original schedules without bandwidth redirection (the preempted receivers will likely recover long before the senders run out of grants on top-priority messages). Thus, when the average length of the preemption intervals increases from 5 to 10 (the degree of overcommitment is also 10 in the experiments), there is a significant bump in the worst-case total buffer occupancy for all message skewness.

Figure 6.18b shows the worst-case queue length of individual output ports at the ToR instead. It's clear that the maximum queue length is bounded by the receiver's degree of overcommitment: at most 9 RTT worth of inbound packets can be queued at the switch (1 RTT worth of packets will be on the wire), and the RTT within a rack is 8 timesteps, so the maximum queue length is $9 \times 8 \times 1500$



(a) Max total buffer occupancy at the ToR.

(b) Max queue length at individual output ports.

Figure 6.18: Worst-case buffer occupancy at the ToR switch as a function of message skewness. The two graphs illustrate the buffer occupancy of the entire switch and of individual output ports, respectively. Each curve denotes a different average length of the preemption intervals, in multiples of the round-trip time. The raw data are extracted from the same simulations as Figure 6.13.

bytes, or 0.108 Mbytes. Assuming 16 MB of total buffer space at the switch, this suggests that it's safe to increase the current overcommitment by almost 4x with no risk of buffer overflow, even if the switch has a fixed-size buffer for each port (as opposed to a dynamic buffer pool that shares buffer space across all ports).

6.3 Limited core bandwidth

When the core bandwidth available in the network is limited, the performance bottleneck of shuffle shifts from the edge of the network to the core (it is no longer possible to shuffle at the line speed of end hosts). Thus, in order to estimate the maximum possible shuffle throughput, one can locate the most loaded network link between the ToR and the core switches, and then divide the amount of data flow through that link by its link speed (recall that the simulation uses an idealized two-level fat tree topology, so there is a unique path between any two nodes in the cluster).

The lower the capacity of the bottleneck core link, or the more data that need to be transferred over it, the lower the maximum throughput that can be achieved in theory. Figure 6.19 shows how these two factors affect the throughput upper bound of a 160-node cluster with 4 racks. First, the throughput upper bound increases linearly with the core bandwidth available. For example, when all the messages have roughly the same size, 75% core bandwidth is just enough to shuffle at 25 Gbps,

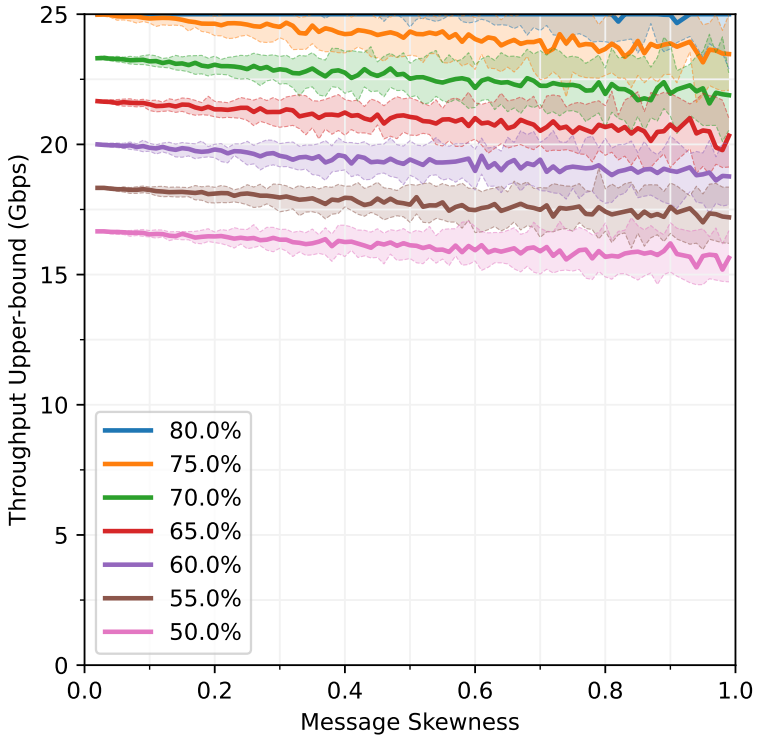


Figure 6.19: Theoretical shuffle throughput upper bound as a function of message skewness. There are a total of 160 nodes, randomly placed in 4 racks (each ToR switch has 40 nodes underneath). Different curves denote different percentages of core bandwidth available: e.g., 50% core bandwidth means the bandwidth between any ToR and the core switch is equal to 50% of the aggregate bandwidth of the end hosts within a rack. Since the workloads are generated randomly (and the nodes are randomly placed in racks), the throughput upper bound varies in different runs. 100 simulation runs are performed for each value of x. The solid lines indicate the median throughput upper bounds, while the shaded regions indicate the 10th and 90th percentiles.

since each rack needs to send (receive) exactly 75% of the outgoing (incoming) data of its end hosts to (from) other racks; as the core bandwidth available drops to 50% (i.e., $\frac{2}{3}$ of 75%), the throughput upper bound also drops to $\frac{2}{3}$ of 25 Gbps, or 16.7 Gbps. Second, as the message skewness increases, the throughput upper bound decreases roughly linearly (both medians and 90th percentiles). This is because the amount of data transferred over the bottleneck core link increases linearly with the message skewness; Figure 6.20 shows the load of the bottleneck core link as a function of the message skewness.

6.3.1 Straggler mitigation extensions

The rest of the section evaluates three versions of the GRPF algorithm that have increasing performance but also come with more restrictions in applicability. The first version, `GRPF+Prio@Edge`, is

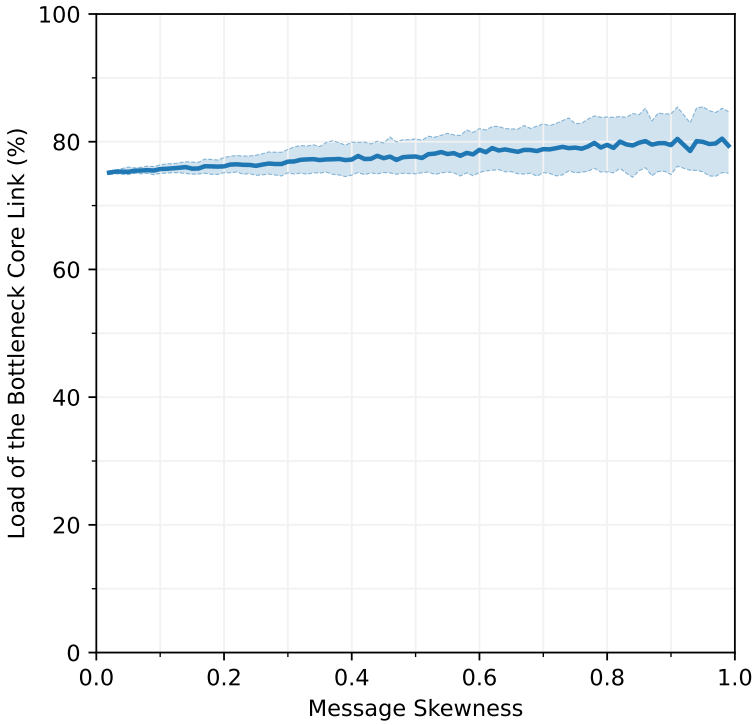
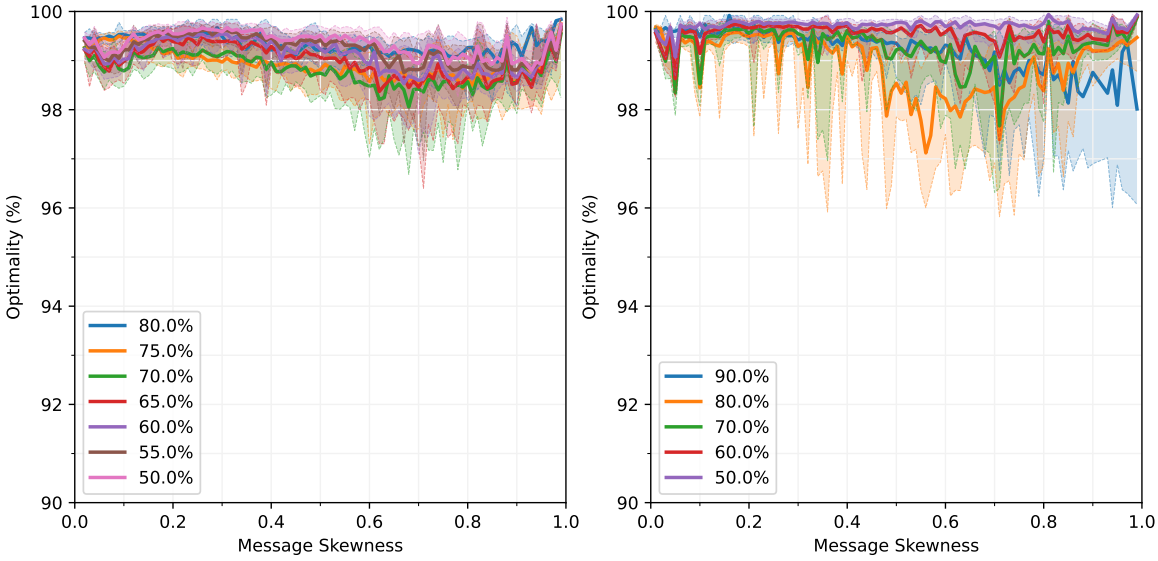


Figure 6.20: Maximum amount of data transferred across the bottleneck core link, in percentage of the average amount of data per rack. The experiment setup is the same as Figure 6.19. The solid line indicates the median load, and the shaded areas indicate the 10th and 90th percentiles.

the default algorithm that has been used in the evaluation so far; in particular, in-network priorities are only enabled on the output ports of the ToR switches that are connected to the end hosts. The second version, `GRPF+Prio@Core`, makes no change to the algorithm itself, but it enables the support for in-network priority at all output ports of all switches (grants will experience no queuing delay). The third and best-performing version, `GRPF+GlobalSched`, is derived from Equation 3.14 (with quadratic scale back) in Section 3.1.7, which further assumes that all nodes will have *instantaneous* access to the number of packets left at the slowest receiver in the cluster. Thus, the performance numbers achieved by this version should be viewed as an upper bound. Fortunately, this assumption can be significantly relaxed in practice: it should be sufficient as long as the latency of the all-reduce operation is relatively low compared to the total shuffle time (discussed later).

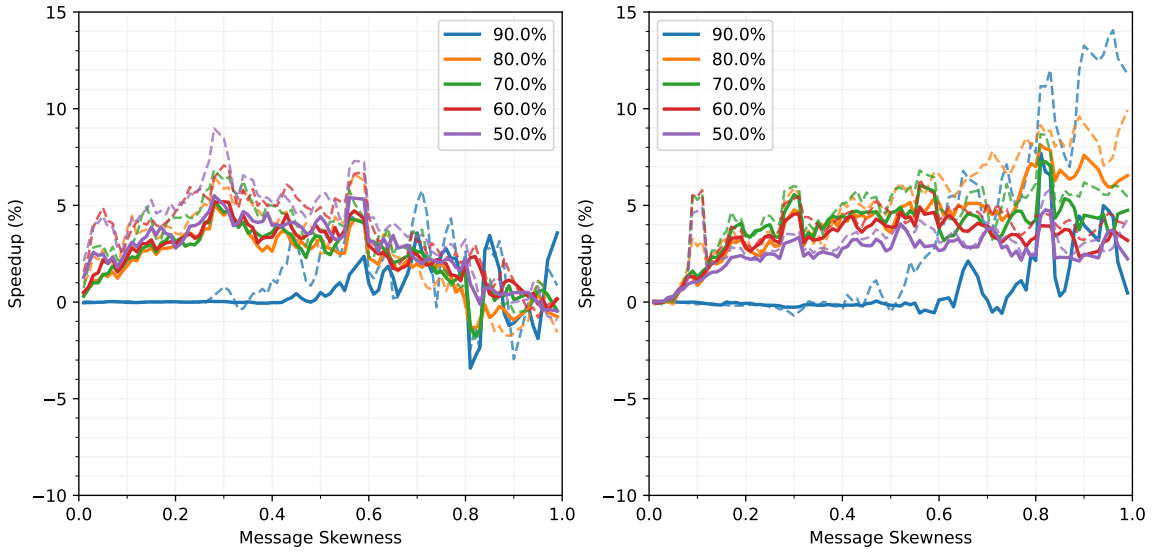
Figure 6.21 shows that `GRPF+GlobalSched` can achieve near-optimal performance, regardless of the message skewness. The performance loss compared to the theoretical upper bound in Figure 6.19 is only about 1% at median, and no more than 4.1% at 90th percentile. However, this algorithm is more limited in applicability compared to the other two versions.

Figure 6.22 evaluates the performance gain achieved by `GRPF+Prio@Core` and `GRPF+GlobalSched`



(a) 4x40 nodes **(b)** 8x40 nodes

Figure 6.21: Optimality of `GRPF+GlobalSched` as a function of message skewness. Optimality is the ratio between the actual throughput and the theoretical upper bound. The two figures simulate a total of 160 and 320 nodes, respectively. The average message size is 16 full packets. Other notations are the same as Figure 6.19. Note that the y-axes are not zero-based.



(a) `GRPF+Prio@Edge` vs. `GRPF+Prio@Core` **(b)** `GRPF+Prio@Core` vs. `GRPF+GlobalSched`

Figure 6.22: Relative speedups achieved by switching from `GRPF+Prio@Edge` to `GRPF+Prio@Core`, and from `GRPF+Prio@Core` to `GRPF+GlobalSched`. Different curves denote different percentages of core bandwidth available. The raw throughputs are taken from the 320-node experiment in Figure 6.21. Solid and dashed lines indicate the speedups of the median and 90th-percentile throughput, respectively.

individually. It focuses on the experiment with 8x40 nodes, as the relative speedups are more significant. Figure 6.22a shows that allowing grants to bypass queues is in general a favorable optimization, especially for low to medium message skewness. At high message skewness, the performance gain drops, and there are even some negative gains at very high message skewness. As discussed in Section 3.1.7, packet priorities eliminate the collateral damage of large packet queues in the network (if a grant is delayed too much, it could result in under-utilization of the uplink of the message sender, or some core link in the path of the message). It appears that, at very high message skewness where each sender has one major outgoing message destined to a distinct receiver, the senders have no trouble saturating all the bottleneck core links even though the grants experience long delays in the network core (due to the lack of in-network priority support).

Figure 6.22b shows that using global information for straggler mitigation consistently improves the performance across the whole range of message skewness. Recall that Section 3.1.7 argued that the root cause of performance degradation is suboptimal bandwidth allocation of the bottleneck core links: i.e., when some messages finish too fast, the remaining messages are not sufficient to saturate the bottleneck core links on their paths. This is supported by the detailed packet traces: when the straggler mitigation mechanism is in place, the messages that pass through the same core link share the link capacity much more closely to an ideal pro rata rate allocation scheme, and the bottleneck core links are kept fully utilized until the very end of shuffle.

Figure 6.23 relaxes the assumption that nodes will have instantaneous accesses to the information of the slowest receiver and investigates how the all-reduce latency impacts the effectiveness of global scheduling. In the best case, an optimal software-based implementation of the all-reduce operation can complete in $\lceil log_2(N) \rceil \times 0.5RTT$, where N is the number of nodes; this is achieved if the network is unloaded or the messages are assigned the highest network priority to bypass the queues. Further, a topology-aware switch-based implementation may complete an all-reduce in 5-10 μs for up to 1000 nodes [84]. In practice, the performance will probably lie between the curves of `Stale1` and `Stale3` (closer to the former), or at most 2-3% worse than `Fresh`.

Finally, Figure 6.24 shows the throughput and optimality of the Hadoop algorithm. Due to the lack of straggler mitigation techniques, its performance is strictly worse than `GRPF+GlobalSched`; in the worst case, it leaves 20–30% of the performance on the table.

I omit the evaluation of the MADD algorithm because its performance will be optimal assuming the underlying network topology is known a priori. However, this assumption rarely holds in practice.

6.3.2 Buffer occupancy

Figure 6.25 shows the average and worst-case buffer utilization at the ToR switches for all three versions of the algorithm. If each ToR switch has 16 MBytes of buffer space (which seems to be a reasonable number for a 25 Gbps ToR switch in practice), then none of the these algorithms is close to getting buffer overflows. Further measurements show that more than 90% of the buffer buildups

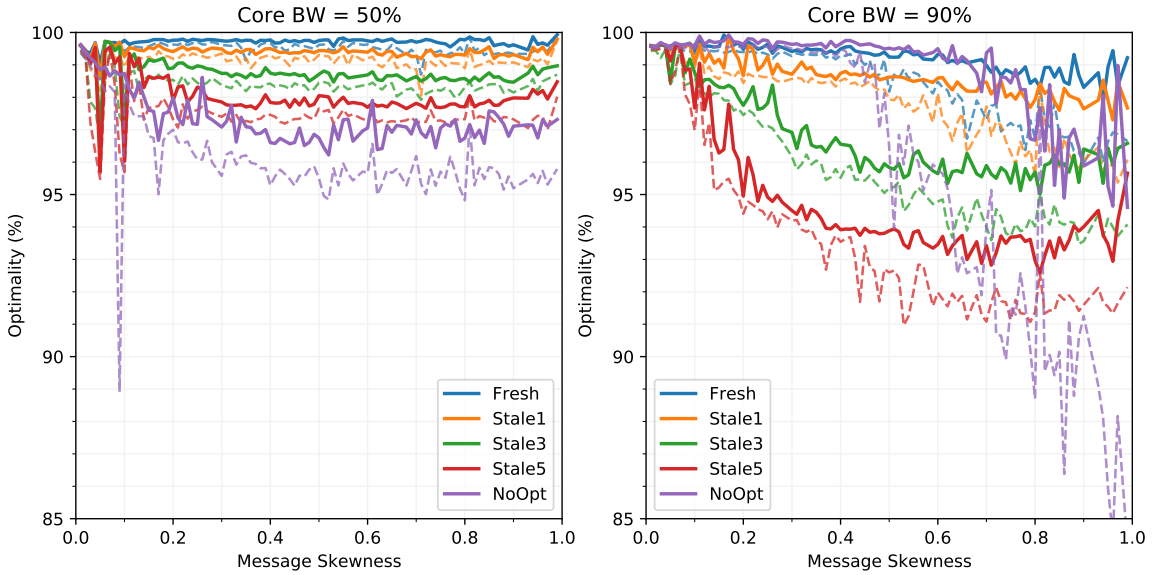


Figure 6.23: The impact of all-reduce latency on shuffle performance. `Fresh` assumes instantaneous accesses to the amount of data left at the slowest receiver (i.e., the same as `GRPF+GlobalSched`). `StaleX` assumes that the all-reduce operation used to exchange such global information will complete X times slower than an optimal software implementation. `NoOpt` is the same as `GRPF+Prio@Core`. The experiment setup is the same as Figure 6.21. Solid and dashed lines indicate the optimality of the median and 90th-percentile throughput, respectively.

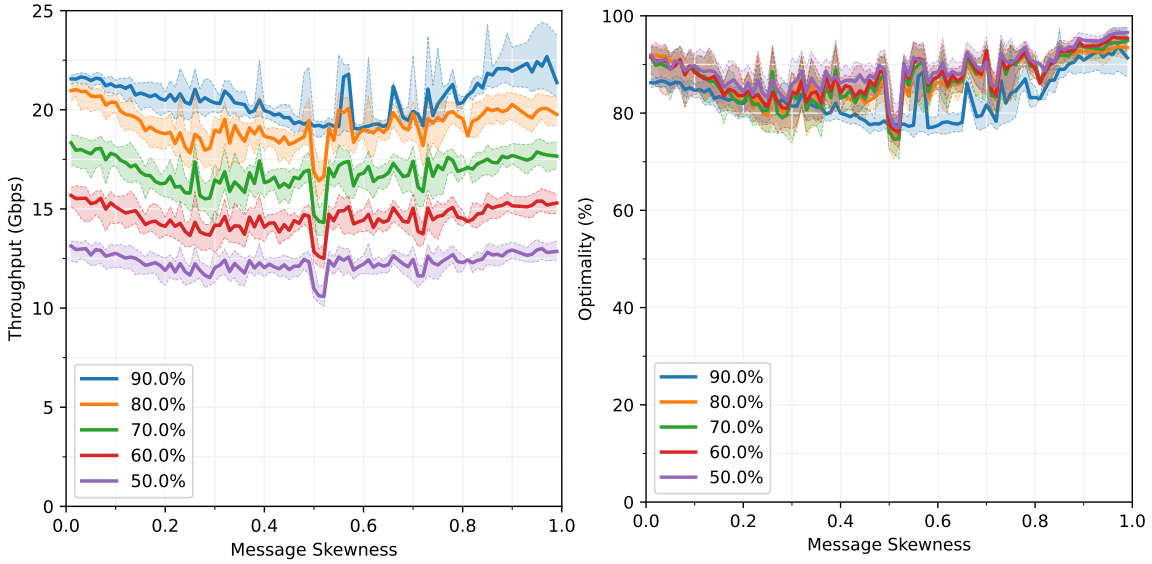
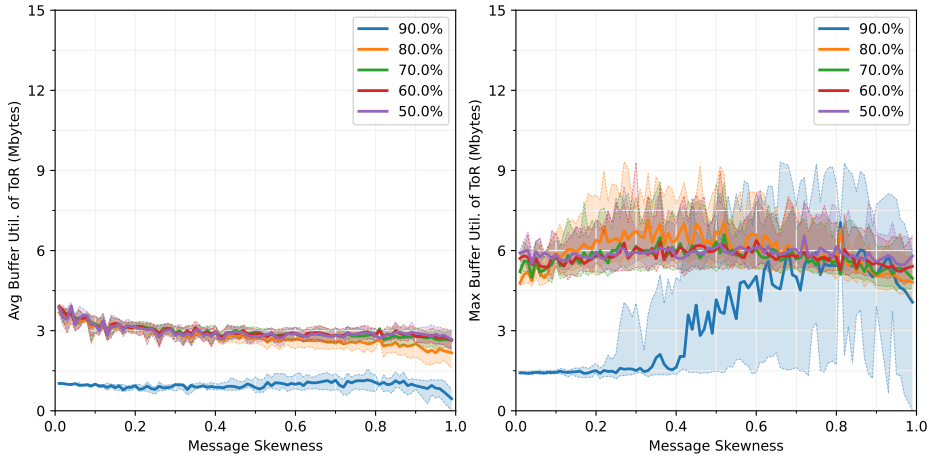
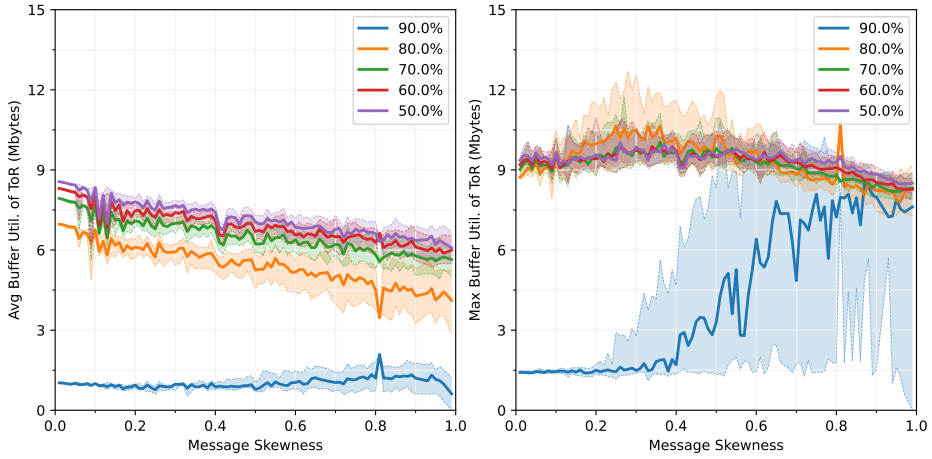


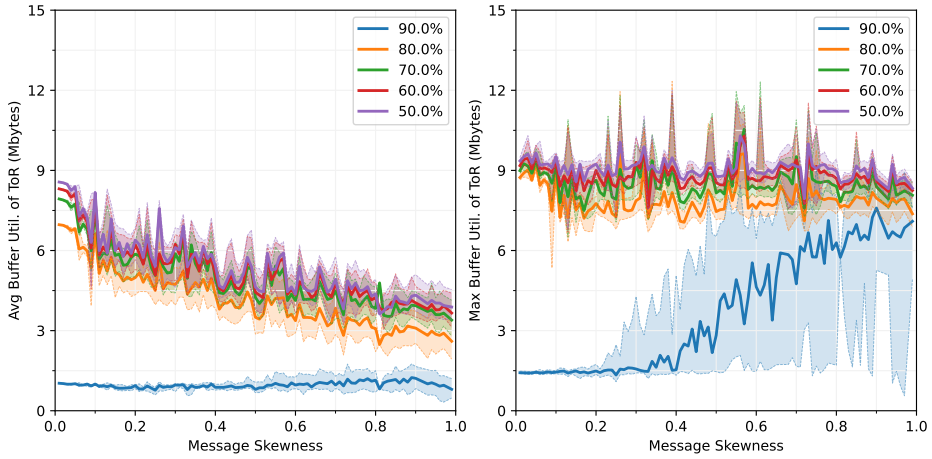
Figure 6.24: Throughput and optimality of the `Hadoop-10` algorithm. Different curves denote different percentages of core bandwidth available. The experiment simulates a total of 8x40 nodes. The average message size is 16 full packets. Other notations are the same as Figure 6.19.



(a) GRPF+Prio@Edge



(b) GRPF+Prio@Core



(c) GRPF+GlobalSched

Figure 6.25: Buffer utilization at ToR's for three versions of the algorithm (320-node experiment).

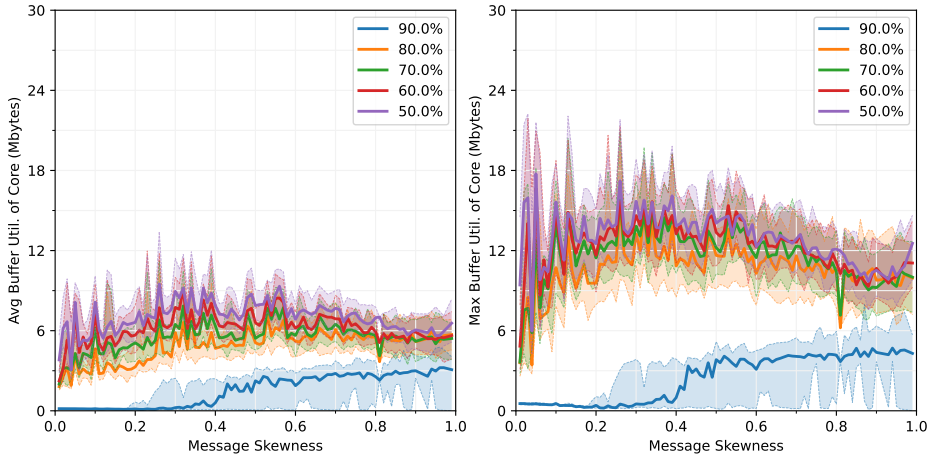
occur at the ToR uplinks to the core switch (as opposed to downlinks of the end hosts). Finally, the curves representing 90% core bandwidth stand out in the graphs because, on average, each rack only needs to send (receive) 87.5% of its data to (from) the other 7 racks (this is the only setup that has sufficient core bandwidth in the experiment).

Figure 6.25 also shows that the buffer occupancy increases significantly when priority is enabled at the network core, and then it drops slightly after the straggler mitigation is in place. The buffer occupancy increases first because much fewer grants will be in flight if they can bypass the queues, and the number of data packets in flight increases as the number of grants in flight drops (the sum of these two is a constant assuming no grant is left unused at the senders). That is, the end hosts can now inject more data packets into the network (although they won't be delivered any sooner). The buffer occupancy later drops in Figure 6.25c because the straggler mitigation mechanism works by reducing the amount of outstanding grants (and, thus, the amount of data that can be injected to the network): more specifically, Equation 3.14 entails that (i) receivers that have fewer remaining bytes, or non-straggler receivers, must issue fewer total outstanding grants, and (ii) messages that are destined to non-straggler receivers have to reduce their sliding window sizes.

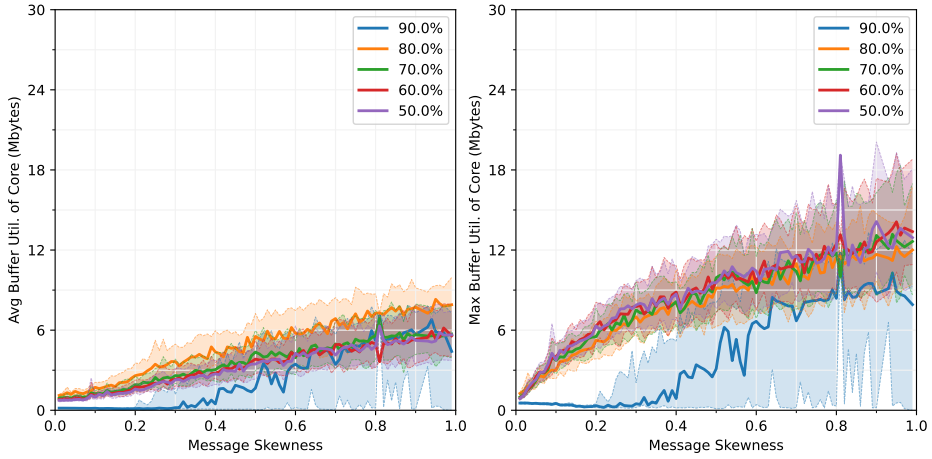
Figure 6.26 shows the utilization of the logical gigantic core switch used in simulation. In practice, the core of the network will consist of multiple aggregation and spine switches, organized in multiple levels, so the buffer occupancy of each individual switch is likely to be lower. For example, if a two-level fat-tree topology with four spine switches (half of the number of ToR switches) is used to provide the 50% core bandwidth in experiments, and the packets are evenly distributed across the spine switches, then the actual buffer occupancy will be only $\frac{1}{4}$ of that in Figure 6.26.

Even though Figure 6.26 shows the total buffer occupancy at the core, its values are not much higher than the buffer occupancy of individual ToR switches in Figure 6.25, which means that buffer overflow will be less of a concern at the core in practice. The buffer occupancy at the core is relatively low compared to ToR switches because buffer buildup at the core can only be caused by ToR fanin (Figure 3.9); the aggregate uplink and downlink bandwidth of the ToR switches are always equal, which makes larger buffer buildups very unlikely unless the workload has a significant incast pattern (also very unlikely given that the workload used in evaluation has uniform data distribution). In fact, the experiments confirm that the most congested output port of the core switch is usually the one connected to the rack that receives the most data from other racks (if the algorithm follows the pro rata rate allocation scheme precisely, this is always true).

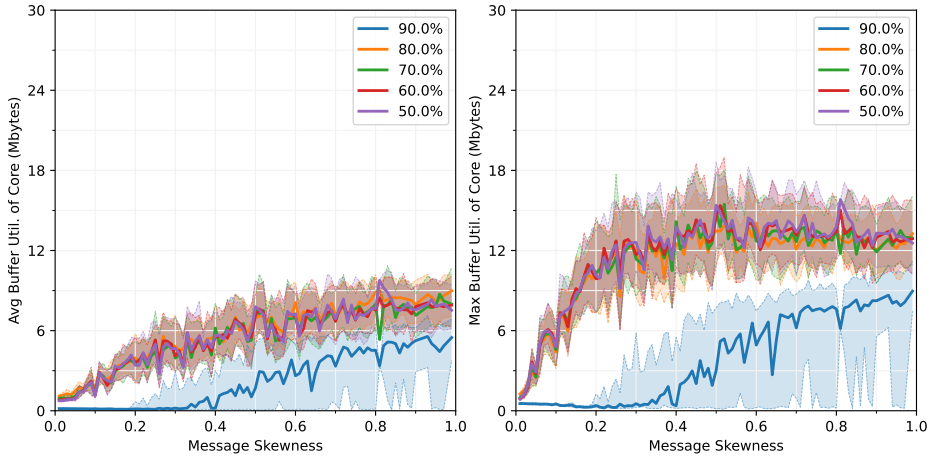
Unlike the buffer occupancy at the edge, the exact relation between the buffer occupancy at the core and the algorithm in use is rather difficult to reason about: it depends on how the ToR switches are using their uplink bandwidth at run time. Further, the run-time behaviors of the ToR's can be affected by the choice of algorithm in intricate and non-deterministic ways. For example, eliminating the queuing delays of the grants increases the buffer occupancy of the ToR's because it allows the end hosts to inject more data packets into the network; however, the buffer occupancy at the core



(a) `GRPF+Prio@Edge`



(b) `GRPF+Prio@Core`



(c) `GRPF+GlobalSched`

Figure 6.26: Buffer utilization at the core for three versions of the algorithm (320-node experiment).

actually drops in many cases, since the larger queues at the ToR's could change the uplink bandwidth allocation in a way that results in a more even distribution of the packets across the destination racks (another potential factor is that senders won't attempt to redirect their bandwidth elsewhere if grants are delivered in time). For another example, straggler mitigation reduces the buffer occupancy at the ToR's by scaling back the sliding windows; however, the buffer occupancy at the core slightly increases instead. Again, varying the sliding windows changes the uplink bandwidth allocation at the ToR's, which further changes the traffic distribution among the racks (more packets from the straggler messages are piled up at the core, which allows them to take a larger share of link capacity). Fortunately, as discussed earlier, buffer overflow is less of a concern at the core, so these issues are secondary at best.

Finally, if the buffer occupancy at the ToR's ever becomes a problem, one can always resort to a smaller degree of overcommitment. Halving the degree of overcommitment also roughly halves the worst-case buffer occupancy at the ToR switches, but it usually only results in very minor drop in performance. For example, when the overcommitment level of `GRPF+GlobalSched` is reduced from 10 to 5, the performance drops by no more than a few percent.

6.4 Summary

Our evaluation shows that the GRPF algorithm can provide near-optimal performance for a variety of workloads, even in the cases of application preemption and limited core bandwidth, and its buffer occupancy is low enough to be free of buffer overflow. It outperforms the Hadoop algorithm in every case. Although the MADD algorithm also achieves the optimal performance in an ideal environment, it requires knowing the underlying network topology a priori, which limits its applicability in practice.

Nonetheless, due to the limitations of the simulator (Section 4.4), the performance results should be taken with a grain of salt. In particular, the simulator chose to model an over-simplified network topology, which cannot reflect the intricate effects of load balancing within the network core. Other factors such as software overheads and noise in the environment may also impact the results. Thus, a proper evaluation using a real environment will be left to future work.

The design features of the GRPF algorithm work coherently to push the performance to the limit. In an ideal environment, the following features are essential to the optimal performance (listed in descending order of importance): (i) the GRPF policy, (ii) overcommitment of receiver downlinks, (iii) pro rata grant window limit, (iv) randomization, and (v) pro rata unscheduled data. Although a small amount of overcommitment is sufficient in an ideal environment, achieving good performance in the presence of application preemption requires a much large degree of overcommitment. Finally, in-network priority is not necessary in an ideal environment, thanks to pro rata grant window limit; however, it is critical to implementing the straggler mitigation extensions of the GRPF algorithm in the case of limited core bandwidth.

Chapter 7

Discussions

This chapter discusses related works and additional topics.

7.1 Shuffles in HPC

Shuffle, or more commonly known as all-to-all communication in the HPC community, is a frequently used communication pattern in many HPC workloads such as fast fourier transform [23], relational algebra [33], graph analytics [32], and molecular simulation [79]. Section 3.1.2 already covers some prior research on the shuffle algorithm conducted within the HPC community, so I will not repeat the discussion here. In general, shuffle algorithms used in HPC often optimize for specific patterns that are common in their applications (e.g., uniform data distribution), and thus their performance will degrade when the workloads become more irregular.

7.2 Shuffles in distributed data-parallel systems

Shuffle is a also key primitive in today's large-scale data-parallel systems such as MapReduce [21] and Spark[100]. However, the shuffle operations in these systems involve more than just all-to-all network communication. For example, a complete shuffle in MapReduce includes the following tasks. First, given M mappers and R reducers, it needs to create $M \times R$ intermediate data blocks (both M and R can be much larger than the number of physical machines). Second, it needs to transfer these intermediate blocks from mappers to reducers, during which time every block needs to be moved across disk, memory, and network at least once. Third, it needs to perform the reduce operation over the input records received from the mappers. Finally, it needs to handle node and network failures that occur in the process and ensures exactly-once semantics (each input record appears exactly once in the output).

As a result, the primary goals of the previous works on shuffles in data-parallel systems [64, 102, 86] are high I/O efficiency and reliability (network efficiency is less important to the overall performance). In contrast, this dissertation assumes that data are already present in memory and only optimizes the all-to-all network communication aspect of the problem (this setup is common in HPC and flash bursts). Still, the GRPF algorithm developed in this dissertation can be applied to these systems should network efficiency ever become a bottleneck.

7.3 Coflow scheduling

So far this dissertation only studies the scheduling of individual messages within a shuffle to minimize the overall completion time. However, another important question is how to schedule different shuffle operations to optimize for metrics such as the average completion time. This problem is also known as *coflow scheduling* in the networking community (coflow is a synonym for collective communication primitive such as broadcast, reduce, or shuffle), and this has generated a rich body of research since its inception [12, 14, 15, 13, 11, 103, 35, 10]. Thus, a potentially interesting future direction could be to explore the combination of the shuffle algorithm and the coflow scheduling algorithm to achieve better overall performance in the presence of concurrent shuffle operations.

7.4 Low-latency network transports

Low-latency transport protocols for datacenter networks have been studied extensively recently, with many protocols proposed to replace TCP in datacenters [69, 36, 55]. These protocols usually aim to achieve low (tail) latency for small messages and high throughput for large messages. In particular, Homa [69] is able to provide near-optimal average and tail message slowdowns even when running at 80% network load (slowdown is the ratio of the actual time required to complete a message divided by the best possible time on an unloaded network). However, these transport protocols only optimize for metrics of individual messages, which is in conflict with the goal of optimal shuffles.

As a result, it's not clear how the shuffle algorithm will be deployed along with these lower-level network transports. One possibility would be to somehow implement the shuffle algorithm on top of these network transports; however, this seems difficult because of the huge differences between their designs (e.g., none of these protocols employs the GRPF policy). And if shuffles must be handled by an entirely different protocol from the network transport, how would they interfere with each other in the presence of mixed workloads? I leave these questions to future work.

Finally, pHost [29] pioneers the idea of expressing flexible networking policies using pull-based end-host scheduling. In particular, it advocates receiver-side ensemble-wise allocation of grants in service of a coflow-wide objective; this approach is then evaluated using a number of policies such as SRPT, Earliest Deadline First, and a multi-tenant sharing policy via round robin. The receiver-side

scheduling of my shuffle protocol shares the same spirit as pHost, with an explicit goal of minimizing the completion time of the last message in the coflow.

7.5 Hardware-assisted network transports

Most transport protocols are implemented in software today. For better performance, several recent systems decide to move the network transports from the kernel to the user space [77, 61, 65, 75, 50]. While these user-space systems reduce the software overheads significantly compared to kernel-based implementations, they still struggle to keep up with high-speed datacenter networks today (e.g., Snap requires 7–14 cores just to drive a 100 Gbps network at 80% load bidirectional [76]). Yet the network speeds are still increasing much faster than CPU speeds. Thus, future network transports will likely need to be offloaded to the NIC to keep up with the ever-increasing line rates [76].

The same problem has been observed in the MilliSort experiment as well. When the total amount of data to sort is fixed, as MilliSort attempts to harness more servers, the amount of data per server drops, and the performance eventually becomes limited by the per-message software overheads in shuffles. By reducing the software overheads of packet processing, it's possible to scale out the sorting algorithm even further: e.g., NanoSort [46] shows an order of magnitude speedup over MilliSort when sorting 1M items by taking advantage a new NIC/CPU hardware design called the nanoPU [41].

Another possibility is to offload the entire collective communication operation to the NICs and switches. This approach provides additional performance benefits, since the switches can use the network topology to optimize the communication: e.g., NVIDIA Scalable Hierarchical Aggregation and Reduction Protocol (SHARP) [34] is capable of offloading several MPI operations to the hardware and effectively reduces the amount of data traversing the network for operations such as broadcast and all-reduce. However, shuffle is not yet supported by SHARP, likely due to the lack of a good shuffle algorithm until now. It remains an interesting question whether the GRPF algorithm can be efficiently implemented in the hardware; this is left to future work.

Chapter 8

Conclusions

In this dissertation, I presented a clean-slate shuffle algorithm that achieves excellent performance for a wide range of workloads and under various conditions such as non-homogeneous ingress/egress rates, interference from competing workloads, and limited bandwidth in the core of the network. At the high level, the new algorithm can also be viewed as an elegant implementation of the more general Weighted Shuffle Scheduling (WSS) scheme [14] in a datacenter environment; it combines a number of simple design features in a synergistic way to achieve this goal:

- **Ensemble-wide scheduling**. Each end host uses a local policy, Greatest Remaining Processing Fraction (GRPF), to schedule its incoming and outgoing messages as ensembles in order to match sender and receiver bandwidths in a decentralized way.
- **Overcommitment.** Keeping a large number of outstanding grants per receiver helps maintain high network utilization during the interference from competing workloads.
- **Pro Rata Sliding Windows.** Sliding windows have been used extensively in window-based flow control schemes; however, instead of using the sliding window to cap the maximum sending rate of a message, my shuffle algorithm uses the ratio between sliding window sizes to enforce the optimal downlink bandwidth allocation among incoming messages (i.e., weighted max-min fair sharing).
- **Lightweight Global Scheduling.** Finally, if nodes are capable of exchanging a small piece of global information periodically with relatively low latency, then an enhanced straggler mitigation technique can be used to further improve the shuffle performance in the case of limited core bandwidth without knowing the underlying network topology a priori.

The evaluation shows that the throughput of the resulting shuffle algorithm is usually within a few percent of the theoretical limit even in the 90th percentile.

This dissertation was strongly motivated by my prior research on flash busrts, a new computing paradigm that emphasizes low-latency data-intensive computation. The measurements of MilliSort and MilliQuery show that shuffle is the critical bottleneck at the optimal operating points of flash bursts. As a result, I expect this new shuffle algorithm to become a critical part for emerging flash burst applications; meanwhile, traditional big-data applications may also adopt this algorithm if the network communication cost of shuffle turns out to be a bottleneck for them.

Bibliography

[1] Alexandru Agache, Marc Brooker, Alexandra Iordache, Anthony Liguori, Rolf Neugebauer, Phil Piwonka, and Diana-Maria Popa. Firecracker: Lightweight virtualization for serverless applications. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*, pages 419–434, 2020.

[2] Anon et al, Dina Bitton, Mark Brown, Rick Catell, Stefano Ceri, Tim Chou, Dave DeWitt, Dieter Gawlick, Hector Garcia-Molina, Bob Good, Jim Gray, Pete Homan, Bob Jolls, Tony Lukes, Ed Lazowska, John Nauman, Mike Pong, Alfred Spector, Kent Trieber, Harald Sammer, Omri Serlin, Mike Stonebraker, Andreas Reuter, and Peter Weinberger. A Measure of Transaction Processing Power. *Datamation*, 31(7):112–118, April 1985.

[3] Lixiang Ao, Liz Izhikevich, Geoffrey M. Voelker, and George Porter. Sprocket: A serverless video processing framework. In *Proceedings of the ACM Symposium on Cloud Computing*, SoCC '18, pages 263–274, New York, NY, USA, 2018. Association for Computing Machinery.

[4] Krste Asanovic, Rastislav Bodik, James Demmel, Tony Keaveny, Kurt Keutzer, John Kubiatowicz, Nelson Morgan, David Patterson, Koushik Sen, John Wawrzynek, David Wessel, and Katherine Yelick. A View of the Parallel Computing Landscape. *Commun. ACM*, 52(10):56–67, October 2009.

[5] AWS Lambda. `https://aws.amazon.com/lambda`.

[6] AWS for the Edge: Bringing data processing and analysis closer to end-points. `https://aws.amazon.com/edge/`.

[7] Michael Axtmann, Sascha Witt, Daniel Ferizovic, and Peter Sanders. In-place Parallel Super Scalar Samplesort (IPSSSSo). *CoRR*, abs/1705.02257, 2017.

[8] Adam Belay, George Prekas, Ana Klimovic, Samuel Grossman, Christos Kozyrakis, and Edouard Bugnion. IX: A Protected Dataplane Operating System for High Throughput and Low Latency. In *Proc. 11th USENIX Conference on Operating Systems Design and Implementation*, OSDI'14, pages 49–65, Berkeley, CA, USA, 2014. USENIX Association.

[9] Dimitri Bertsekas and Robert Gallager. *Data networks*. Athena Scientific, 2021.

[10] Mosharaf Chowdhury, Samir Khuller, Manish Purohit, Sheng Yang, and Jie You. Near optimal coflow scheduling in networks. In *The 31st ACM Symposium on Parallelism in Algorithms and Architectures*, pages 123–134, 2019.

[11] Mosharaf Chowdhury, Zhenhua Liu, Ali Ghodsi, and Ion Stoica. {HUG}:{Multi-Resource} fairness for correlated and elastic demands. In *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16)*, pages 407–424, 2016.

[12] Mosharaf Chowdhury and Ion Stoica. Coflow: A networking abstraction for cluster applications. In *Proceedings of the 11th ACM Workshop on Hot Topics in Networks*, pages 31–36, 2012.

[13] Mosharaf Chowdhury and Ion Stoica. Efficient coflow scheduling without prior knowledge. *ACM SIGCOMM Computer Communication Review*, 45(4):393–406, 2015.

[14] Mosharaf Chowdhury, Matei Zaharia, Justin Ma, Michael I Jordan, and Ion Stoica. Managing data transfers in computer clusters with orchestra. *ACM SIGCOMM computer communication review*, 41(4):98–109, 2011.

[15] Mosharaf Chowdhury, Yuan Zhong, and Ion Stoica. Efficient coflow scheduling with varys. In *Proceedings of the 2014 ACM conference on SIGCOMM*, pages 443–454, 2014.

[16] Jon Crowcroft and Philippe Oechslin. Differentiated end-to-end internet services using a weighted proportional fair sharing tcp. *ACM SIGCOMM Computer Communication Review*, 28(3):53–69, 1998.

[17] Emilie Danna, Avinatan Hassidim, Haim Kaplan, Alok Kumar, Yishay Mansour, Danny Raz, and Michal Segalov. Upward max-min fairness. *Journal of the ACM (JACM)*, 64(1):1–24, 2017.

[18] Emilie Danna, Subhasree Mandal, and Arjun Singh. A practical algorithm for balancing the max-min fairness and throughput objectives in traffic engineering. In *2012 Proceedings IEEE INFOCOM*, pages 846–854. IEEE, 2012.

[19] Frederica Darema. The spmd model: Past, present and future. In *European Parallel Virtual Machine/Message Passing Interface Users' Group Meeting*, pages 1–1. Springer, 2001.

[20] Data Plane Development Kit. `http://dpdk.org/`.

[21] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.

[22] Advait Dixit, Pawan Prakash, Y Charlie Hu, and Ramana Rao Kompella. On the impact of packet spraying in data center networks. In *2013 Proceedings IEEE INFOCOM*, pages 2130–2138. IEEE, 2013.

[23] Jun Doi and Yasushi Negishi. Overlapping methods of all-to-all communication and fft algorithms for torus-connected massively parallel supercomputers. In *SC'10: Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–9. IEEE, 2010.

[24] Aleksandar Dragojević, Dushyanth Narayanan, Miguel Castro, and Orion Hodson. Farm: Fast remote memory. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*, pages 401–414, Seattle, WA, April 2014. USENIX Association.

[25] Daniel Firestone, Andrew Putnam, Sambhrama Mundkur, Derek Chiou, Alireza Dabagh, Mike Andrewartha, Hari Angepat, Vivek Bhanu, Adrian Caulfield, Eric Chung, et al. Azure accelerated networking: Smartnics in the public cloud. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*, pages 51–66, 2018.

[26] Apache Flink - Stateful Computations over Data Streams, Jan 2021. `https://flink.apache.org`.

[27] Sadjad Fouladi, Riad S. Wahby, Brennan Shacklett, Karthikeyan Vasuki Balasubramaniam, William Zeng, Rahul Bhalerao, Anirudh Sivaraman, George Porter, and Keith Winstein. Encoding, fast and slow: Low-latency video processing using thousands of tiny threads. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI '17)*, Boston, Massachusetts, USA, 2017. USENIX.

[28] Joshua Fried, Zhenyuan Ruan, Amy Ousterhout, and Adam Belay. Caladan: Mitigating interference at microsecond timescales. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. USENIX Association, November 2020.

[29] Peter X Gao, Akshay Narayan, Gautam Kumar, Rachit Agarwal, Sylvia Ratnasamy, and Scott Shenker. phost: Distributed near-optimal datacenter transport over commodity network fabric. In *Proceedings of the 11th ACM Conference on Emerging Networking Experiments and Technologies*, pages 1–12, 2015.

[30] Wanling Gao, Jianfeng Zhan, Lei Wang, Chunjie Luo, Daoyi Zheng, Fei Tang, Biwei Xie, Chen Zheng, Xu Wen, Xiwen He, et al. Data Motifs: A Lens Towards Fully Understanding Big Data and AI Workloads. In *Proceedings of the 27th International Conference on Parallel Architectures and Compilation Techniques*, pages 1–14, 2018.

[31] Google Cloud Platform. `https://cloud.google.com/`.

[32] Sayan Ghosh, Mahantesh Halappanavar, Ananth Kalyanaraman, Arif Khan, and Assefaw H Gebremedhin. Exploring mpi communication models for graph applications using graph matching as a case study. In *2019 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 761–770. IEEE, 2019.

[33] Thomas Gilray and Sidharth Kumar. Distributed relational algebra at scale. In *2019 IEEE 26th International Conference on High Performance Computing, Data, and Analytics (HiPC)*, pages 12–22. IEEE, 2019.

[34] Richard L Graham, Devendar Bureddy, Pak Lui, Hal Rosenstock, Gilad Shainer, Gil Bloch, Dror Goldenerg, Mike Dubman, Sasha Kotchubievsky, Vladimir Koushnir, et al. Scalable hierarchical aggregation protocol (sharp): a hardware architecture for efficient data reduction. In *2016 First International Workshop on Communication Optimizations in HPC (COMHPC)*, pages 1–10. IEEE, 2016.

[35] Robert Grandl, Mosharaf Chowdhury, Aditya Akella, and Ganesh Ananthanarayanan. Altruistic scheduling in {Multi-Resource} clusters. In *12th USENIX symposium on operating systems design and implementation (OSDI 16)*, pages 65–80, 2016.

[36] Mark Handley, Costin Raiciu, Alexandru Agache, Andrei Voinescu, Andrew W Moore, Gianni Antichi, and Marcin Wójcik. Re-architecting datacenter networks and stacks for low latency and high performance. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, pages 29–42, 2017.

[37] Vipul Harsh, Laxmikant Kale, and Edgar Solomonik. Histogram sort with sampling. In *The 31st ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '19, page 201–212, New York, NY, USA, 2019. Association for Computing Machinery.

[38] Joel Hasbrouck and Gideon Saar. Low-latency trading. *Journal of Financial Markets*, 16(4):646–679, 2013.

[39] Michael Hofmann, Gudula Rünger, Paul Gibbon, and Robert Speck. Parallel sorting algorithms for optimizing particle simulations. In *2010 IEEE International Conference On Cluster Computing Workshops and Posters (CLUSTER WORKSHOPS)*, pages 1–8. IEEE, 2010.

[40] Justin Hu, Ariana Bruno, Drew Zagieboylo, Mark Zhao, Brian Ritchken, Brendon Jackson, Joo Yeon Chae, Francois Mertil, Mateo Espinosa, and Christina Delimitrou. To centralize or not to centralize: A tale of swarm coordination. *arXiv preprint arXiv:1805.01786*, 2018.

[41] Stephen Ibanez, Alex Mallery, Serhat Arslan, Theo Jepsen, Muhammad Shahbaz, Changhoon Kim, and Nick McKeown. The nanopu: A nanosecond network stack for datacenters. In *15th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 21)*, pages 239–256, 2021.

[42] Daos: Distributed asynchronous object storage, August 2020. `https://github.com/daos-stack/daos`.

[43] Intel OPA-PSM2 Git Repository, February 2020. `https://github.com/intel/opa-psm2.git`.

[44] Călin Iorgulescu, Reza Azimi, Youngjin Kwon, Sameh Elnikety, Manoj Syamala, Vivek Narasayya, Herodotos Herodotou, Paulo Tomita, Alex Chen, Jack Zhang, et al. {PerfIso}: Performance isolation for commercial {Latency-Sensitive} services. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, pages 519–532, 2018.

[45] Joseph Izraelevitz, Jian Yang, Lu Zhang, Juno Kim, Xiao Liu, Amirsaman Memaripour, Yun Joon Soh, Zixuan Wang, Yi Xu, Subramanya R. Dulloor, Jishen Zhao, and Steven Swanson. Basic Performance Measurements of the Intel Optane DC Persistent Memory Module, 2019.

[46] Theo Jepsen, Stephen Ibanez, Gregory Valiant, and Nick McKeown. From sand to flour: The next leap in granular computing with nanosort. *arXiv preprint arXiv:2204.12615*, 2022.

[47] Jie Jiang, Lixiong Zheng, Junfeng Pu, Xiong Cheng, Chongqing Zhao, Mark R Nutter, and Jeremy D Schaub. Tencent sort.

[48] Kostis Kaffes, Timothy Chong, Jack Tigar Humphries, Adam Belay, David Mazières, and Christos Kozyrakis. Shinjuku: Preemptive Scheduling for μsecond-scale Tail Latency. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*, pages 345–360, Boston, MA, 2019. USENIX Association.

[49] Kostis Kaffes, Neeraja J Yadwadkar, and Christos Kozyrakis. Centralized core-granular scheduling for serverless functions. In *Proceedings of the ACM Symposium on Cloud Computing*, pages 158–164, 2019.

[50] Anuj Kalia, Michael Kaminsky, and David Andersen. Datacenter RPCs can be General and Fast. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*, pages 1–16, Boston, MA, 2019. USENIX Association.

[51] Erich L Kaltofen. The "seven dwarfs" of symbolic computation. In *Numerical and symbolic scientific computing*, pages 95–104. Springer, 2012.

[52] Changkyu Kim, Jongsoo Park, Nadathur Satish, Hongrae Lee, Pradeep Dubey, and Jatin Chhugani. CloudRAMSort: Fast and Efficient Large-Scale Distributed RAM Sort on Shared-Nothing Cluster. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2012, Scottsdale, AZ, USA, May 20-24, 2012*, pages 841–850, 2012.

[53] R. Kowalewski, P. Jungblut, and K. Fürlinger. Engineering a distributed histogram sort. In *2019 IEEE International Conference on Cluster Computing (CLUSTER)*, pages 1–11, 2019.

[54] Chinmay Kulkarni, Sara Moore, Mazhar Naqvi, Tian Zhang, Robert Ricci, and Ryan Stutsman. Splinter: Bare-metal extensions for multi-tenant low-latency storage. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 627–643, 2018.

[55] Gautam Kumar, Nandita Dukkipati, Keon Jang, Hassan MG Wassel, Xian Wu, Behnam Montazeri, Yaogong Wang, Kevin Springborn, Christopher Alfeld, Michael Ryan, et al. Swift: Delay is simple and effective for congestion control in the datacenter. In *Proceedings of the Annual conference of the ACM Special Interest Group on Data Communication on the applications, technologies, architectures, and protocols for computer communication*, pages 514–528, 2020.

[56] Valliappa Lakshmanan and Jordan Tigani. *Google BigQuery: the Definitive Guide: Data Warehousing, Analytics, and Machine Learning at Scale*. O'Reilly Media, 2019.

[57] Vinh The Lam, Sivasankar Radhakrishnan, Rong Pan, Amin Vahdat, and George Varghese. Netshare and stochastic netshare: predictable bandwidth allocation for data centers. *ACM SIGCOMM Computer Communication Review*, 42(3):5–11, 2012.

[58] Collin Lee and John Ousterhout. Granular computing. In *Proceedings of the Workshop on Hot Topics in Operating Systems*, pages 149–154, 2019.

[59] Xiaobo Li, Paul Lu, Jonathan Schaeffer, John Shillington, Pok Sze Wong, and Hanmao Shi. On the versatility of parallel sorting by regular sampling. *Parallel Computing*, 19(10):1079 – 1103, 1993.

[60] Yuliang Li, Rui Miao, Hongqiang Harry Liu, Yan Zhuang, Fei Feng, Lingbo Tang, Zheng Cao, Ming Zhang, Frank Kelly, Mohammad Alizadeh, and Minlan Yu. Hpcc: High precision congestion control. In *Proceedings of the ACM Special Interest Group on Data Communication*, SIGCOMM '19, page 44–58, New York, NY, USA, 2019. Association for Computing Machinery.

[61] Hyeontaek Lim, Dongsu Han, David G. Andersen, and Michael Kaminsky. Mica: A holistic approach to fast in-memory key-value storage. In *Proc. 11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*, pages 429–444, Seattle, WA, April 2014. USENIX Association.

[62] Shih-Chieh Lin, Yunqi Zhang, Chang-Hong Hsu, Matt Skach, Md E Haque, Lingjia Tang, and Jason Mars. The architectural implications of autonomous driving: Constraints and acceleration. In *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 751–766, 2018.

[63] Shiyu Liu, Ahmad Ghalayini, Mohammad Alizadeh, Balaji Prabhakar, Mendel Rosenblum, and Anirudh Sivaraman. Breaking the {Transience-Equilibrium} nexus: A new approach to datacenter packet transport. In *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*, pages 47–63, 2021.

[64] Frank Sifei Luan, Stephanie Wang, Samyukta Yagati, Sean Kim, Kenneth Lien, SangBin Cho, Eric Liang, and Ion Stoica. Exoshuffle: Large-scale shuffle at the application level. *arXiv preprint arXiv:2203.05072*, 2022.

[65] Michael Marty, Marc de Kruijf, Jacob Adriaens, Christopher Alfeld, Sean Bauer, Carlo Contavalli, Michael Dalton, Nandita Dukkipati, William C Evans, Steve Gribble, et al. Snap: A microkernel approach to host networking. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, pages 399–413, 2019.

[66] Frank McSherry. Graph analysis and hilbert space-filling curves. `https://bigdataatsvc.wordpress.com/2013/07/02/graph-analysis-and-hilbert-space-filling-curves/`, Jul 2013. Accessed: 2020-01-30.

[67] Microsoft Azure Functions. `https://azure.microsoft.com/en-us/services/functions/`.

[68] Radhika Mittal, Vinh The Lam, Nandita Dukkipati, Emily Blem, Hassan Wassel, Monia Ghobadi, Amin Vahdat, Yaogong Wang, David Wetherall, and David Zats. Timely: Rtt-based congestion control for the datacenter. *ACM SIGCOMM Computer Communication Review*, 45(4):537–550, 2015.

[69] Behnam Montazeri, Yilong Li, Mohammad Alizadeh, and John Ousterhout. Homa: A receiver-driven low-latency transport protocol using network priorities. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*, pages 221–235, 2018.

[70] Kanthi Nagaraj, Dinesh Bharadia, Hongzi Mao, Sandeep Chinchali, Mohammad Alizadeh, and Sachin Katti. Numfabric: Fast and flexible bandwidth allocation in datacenters. In *Proceedings of the 2016 ACM SIGCOMM Conference*, pages 188–201, 2016.

[71] ns-2 Main Page. `http://nsnam.sourceforge.net/wiki/index.php/Main_Page`.

[72] NVIDIA. NVIDIA Collective Communications Library (NCCL). `https://developer.nvidia.com/nccl`, 2022.

[73] OMNeT++ Discrete Event Simulator. `http://https://omnetpp.org/`.

[74] Dispersive Routing - Intel Omni-Path Fabric Performance Tuning, Aug 2020. `https://edc.intel.com/content/www/xl/es/design/products-and-solutions/networking-and-io/fabric-products/omni-path/intel-omni-path-performance-tuning-user-guide/dispersive-routing/`.

[75] Amy Ousterhout, Joshua Fried, Jonathan Behrens, Adam Belay, and Hari Balakrishnan. Shenango: Achieving High CPU Efficiency for Latency-sensitive Datacenter Workloads. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*, pages 361–378, Boston, MA, 2019. USENIX Association.

[76] John Ousterhout. A linux kernel implementation of the homa transport protocol. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)*, pages 99–115, 2021.

[77] John Ousterhout, Arjun Gopalan, Ashish Gupta, Ankita Kejriwal, Collin Lee, Behnam Montazeri, Diego Ongaro, Seo Jin Park, Henry Qin, Mendel Rosenblum, Stephen Rumble, Ryan Stutsman, and Stephen Yang. The RAMCloud Storage System. *ACM Trans. Comput. Syst.*, 33(3):7:1–7:55, August 2015.

[78] Jonathan Perry, Amy Ousterhout, Hari Balakrishnan, Devavrat Shah, and Hans Fugal. Fastpass: A centralized" zero-queue" datacenter network. In *Proceedings of the 2014 ACM conference on SIGCOMM*, pages 307–318, 2014.

[79] James C Phillips, Gengbin Zheng, Sameer Kumar, and Laxmikant V Kalé. Namd: Biomolecular simulation on thousands of processors. In *SC'02: Proceedings of the 2002 ACM/IEEE conference on Supercomputing*, pages 36–36. IEEE, 2002.

[80] Balaji Prabhakar. Time perimeters and stock exchanges in the cloud. `https://conferences.sigcomm.org/sigcomm/2020/tutorial-netfinance.html`, Aug 2020. Accessed: 2020-08-30.

[81] George Prekas, Marios Kogias, and Edouard Bugnion. Zygos: Achieving low tail latency for microsecond-scale networked tasks. In *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP '17, pages 325–341, New York, NY, USA, 2017. ACM.

[82] Bogdan Prisacari, German Rodriguez, Cyriel Minkenberg, and Torsten Hoefler. Bandwidth-optimal all-to-all exchanges in fat tree networks. In *Proceedings of the 27th international ACM conference on International conference on supercomputing*, pages 139–148, 2013.

[83] Henry Qin, Qian Li, Jacqueline Speiser, Peter Kraft, and John Ousterhout. Arachne: Core-aware Thread Management. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*, OSDI'18, pages 145–160, Berkeley, CA, USA, 2018. USENIX Association.

[84] Bharath Ramesh, Kaushik Kandadi Suresh, Nick Sarkauskas, Mohammadreza Bayatpour, Jahanzeb Maqbool Hashmi, Hari Subramoni, and Dhabaleswar K Panda. Scalable mpi collectives using sharp: large scale performance evaluation on the tacc frontera system. In *2020 Workshop on Exascale MPI (ExaMPI)*, pages 11–20. IEEE, 2020.

[85] Martin Ruefenacht, Mark Bull, and Stephen Booth. Generalisation of recursive doubling for allreduce: Now with simulation. *Parallel Computing*, 69:24–44, 2017.

[86] Min Shen, Ye Zhou, and Chandni Singh. Magnet: push-based shuffle service for large-scale data processing. *Proceedings of the VLDB Endowment*, 13(12):3382–3395, 2020.

[87] Hanmao Shi and Jonathan Schaeffer. Parallel sorting by regular sampling. *Journal of Parallel and Distributed Computing*, 14(4):361 – 372, 1992.

[88] Alan Shieh, Srikanth Kandula, Albert Greenberg, Changhoon Kim, and Bikas Saha. Sharing the data center network. In *8th USENIX Symposium on Networked Systems Design and Implementation (NSDI 11)*, 2011.

[89] Arjun Singh, Joon Ong, Amit Agarwal, Glen Anderson, Ashby Armistead, Roy Bannon, Seb Boving, Gaurav Desai, Bob Felderman, Paulie Germano, et al. Jupiter rising: A decade of clos topologies and centralized control in google's datacenter network. *ACM SIGCOMM computer communication review*, 45(4):183–197, 2015.

[90] Marc Snir, Steve Otto, Steven Huss-Lederman, David Walker, and Jack Dongarra. *MPI-The Complete Reference, Volume 1: The MPI Core*. MIT Press, Cambridge, MA, USA, 2nd. (revised) edition, 1998.

[91] Sort Benchmark Home Page. `http://sortbenchmark.org`.

[92] Hari Subramoni, Krishna Kandalla, Jithin Jose, Karen Tomko, Karl Schulz, Dmitry Pekurovsky, and Dhabaleswar K Panda. Designing topology-aware communication schedules for alltoall operations in large infiniband clusters. In *2014 43rd International Conference on Parallel Processing*, pages 231–240. IEEE, 2014.

[93] Rajeev Thakur, Rolf Rabenseifner, and William Gropp. Optimization of collective communication operations in mpich. *The International Journal of High Performance Computing Applications*, 19(1):49–66, 2005.

[94] Rajeev Thakur, Rolf Rabenseifner, and William Gropp. Optimization of collective communication operations in mpich. *The International Journal of High Performance Computing Applications*, 19(1):49–66, 2005.

[95] Jordan Tigani and Siddartha Naidu. *Google BigQuery Analytics*. John Wiley & Sons, 2014.

[96] Broadcom's Trident 3 enhances ECMP with Dynamic Load Balancing, Sept 2017. `https://www.broadcom.com/blog/broadcom-s-trident-3-enhances-ecmp-with-dynamic-load-balancing`.

[97] Udayanga Wickramasinghe and Andrew Lumsdaine. A Survey of Methods for Collective Communication Optimization and Tuning. *CoRR*, abs/1611.06334, 2016.

[98] Udayanga Wickramasinghe and Andrew Lumsdaine. A survey of methods for collective communication optimization and tuning. *arXiv preprint arXiv:1611.06334*, 2016.

[99] Wikipedia contributors. Fat tree — Wikipedia, the free encyclopedia. `https://en.wikipedia.org/w/index.php?title=Fat_tree&oldid=1071181144`, 2022. [Online; accessed 2-March-2022].

[100] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauly, Michael J Franklin, Scott Shenker, and Ion Stoica. Resilient distributed datasets: A {Fault-Tolerant} abstraction for {In-Memory} cluster computing. In *9th USENIX Symposium on Networked Systems Design and Implementation (NSDI 12)*, pages 15–28, 2012.

[101] Matei Zaharia, Tathagata Das, Haoyuan Li, Timothy Hunter, Scott Shenker, and Ion Stoica. Discretized streams: Fault-tolerant streaming computation at scale. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, pages 423–438, New York, NY, USA, 2013. ACM.

[102] Haoyu Zhang, Brian Cho, Ergin Seyfe, Avery Ching, and Michael J Freedman. Riffle: optimized shuffle service for large-scale data analytics. In *Proceedings of the Thirteenth EuroSys Conference*, pages 1–15, 2018.

[103] Hong Zhang, Li Chen, Bairen Yi, Kai Chen, Mosharaf Chowdhury, and Yanhui Geng. Coda: Toward automatically identifying and scheduling coflows in the dark. In *Proceedings of the 2016 ACM SIGCOMM Conference*, pages 160–173, 2016.

[104] Qiao Zhang, Vincent Liu, Hongyi Zeng, and Arvind Krishnamurthy. High-resolution measurement of data center microbursts. In *Proceedings of the 2017 Internet Measurement Conference*, pages 78–85, 2017.

[105] Xiao Zhang, Eric Tune, Robert Hagmann, Rohit Jnagal, Vrigo Gokhale, and John Wilkes. Cpi2: Cpu performance isolation for shared compute clusters. In *Proceedings of the 8th ACM European Conference on Computer Systems*, pages 379–391, 2013.

[106] Siyuan Zhuang, Zhuohan Li, Danyang Zhuo, Stephanie Wang, Eric Liang, Robert Nishihara, Philipp Moritz, and Ion Stoica. Hoplite: efficient and fault-tolerant collective communication for task-based distributed systems. In *Proceedings of the 2021 ACM SIGCOMM 2021 Conference*, pages 641–656, 2021.